# An Integer Programming Approach to Optimal Basic Block Instruction Scheduling for Single-Issue Processors

Michael Jünger and Sven Mallach

Institut für Informatik
Universität zu Köln, 50969 Köln, Germany

27th March 2015.

**Abstract**

We present a novel integer programming formulation for basic block instruction scheduling on single-issue processors. The problem can be considered as a very general sequential task scheduling problem with delayed precedence-constraints. Our model is based on the linear ordering problem and has, in contrast to the last IP model proposed, numbers of variables and constraints that are strongly polynomial in the instance size. Combined with improved preprocessing techniques and given a time limit of ten minutes of CPU and system time, our branch-and-cut implementation is capable to solve all but eleven of the 369,861 basic blocks of the SPEC 2000 integer and floating point benchmarks to proven optimality. This is competitive to the current state-of-the art constraint programming approach that has also been evaluated on this test suite.

## 1 Introduction

Today most computer programs are written in high-level programming languages and developers rely on compilers in order to generate executable machine code for various operating systems and processor architectures. One of the fundamental subroutines of any compiler is the instruction scheduling phase where the generated machine instructions shall be ordered such that the number of processor clock cycles needed to complete all the operations is minimized. Modern processor architectures are pipelined, i.e., the execution of a single machine instruction is partitioned into several stages. As a result, multiple instructions can be in flight, occupying different stages at the same time [1]. However, in practice, the ideal flow of instructions through the pipeline may be disturbed by several conflicts, especially such caused by data dependencies between the instructions. Therefore, each precedence relationship has an associated *latency*, capturing the number of clock cycles needed until the result computed by the first instruction is available to its successor. The starting times of dependent instructions must obey these latencies to ensure that no conflicts occur and all operands are present in logic when they execute. These *precedence* and

*latency constraints* make instruction scheduling an $\mathcal{NP}$-hard combinatorial optimization problem [2], even for *single-issue* processors that allow only at most one instruction to be inserted into the pipeline (*issued*) in every clock cycle. Polynomial-time solvability is known only for the very restrictive case that the maximum occurring latency is one clock cycle [3].

In this article, we focus on the exact solution of the basic block instruction scheduling problem for single-issue processors using integer programming. A *basic block* is a series of instructions without any internal branches, i.e., all the instructions need to be scheduled as a straight-line sequence after entering the basic block and before exiting it again. Although more global approaches to instruction scheduling exist, the decomposition of a program into its basic blocks is still a common approach [4]. This is true for many reasons. The consideration of so-called *superblocks*, i.e., a series of basic blocks that allow for side exits, introduces probabilities into the problem since it is unknown whether the branch instruction causing a side exit will be taken at runtime. Further, the opportunity to move instructions across basic block boundaries vastly enlarges the search space whereas basic block scheduling is already a very hard problem with a lot of symmetries.

Production compilers typically rely on a *list scheduling* heuristic, a method maintaining a list of instructions ready to be scheduled and selecting at each clock cycle a ready instruction with the highest among some pre-determined priorities. List scheduling is flexible since it can easily handle parallel or multiple-issue processors and different priority functions. For the problem under consideration, Bernstein, Rodeh, and Gertner [5] showed that, regardless how priorities are assigned, list schedules are no worse than $2 - \frac{1}{L+1}$ times the optimum where $L$ is the maximum latency occurring. Many computational experiments, e.g. [6, 7, 8, 9], reveal near-optimal performance of list scheduling when averaging results over a particular set of instances. The fine-grained results in [7, 8] show however, that the number of basic blocks where list scheduling does not find optimal schedules grows significantly with increasing size of the instances (up to 20% for basic blocks with more than 250 instructions in their results). Moreover, provably optimal schedulers are desirable to enable quality measures but also in settings where the runtime performance of the final programs is critical or where longer compile times are tolerable. This is the case, e.g., for embedded and digital signal processing applications or, in general, software that is pre-compiled only once before (mass) delivery.

Early branch-and-bound [10, 11], integer programming (IP) [12, 13, 14] and constraint programming (CP) [6] approaches were limited to small sets of instances with roughly up to 50 instructions. The most recent contribution to attack the instruction scheduling problem with integer programming was given by Wilken, Liu and Heffernan [9] in 2000. They were the first to optimally schedule a larger set of basic blocks with up to $1,000$ instructions by applying some search space reduction techniques and problem-specific cutting plane separation. However, their experiments were restricted to instances with latencies in the range between zero and two clock cycles and later it was then shown that the method is not as successful on more realistic instances with larger and varying latencies [15]. The method has also a limited scalability since the number of variables and constraints is pseudo-polynomial in the size of the input, it depends on (an upper bound on) the makespan that can be much larger than the number of instructions. An important result of their work is however that the reduction

techniques are essential to be able to schedule real-world instances to optimality. One year later, van Beek and Wilken [16] proposed a constraint programming approach that could optimally schedule the instances used for the experiments by Wilken, Liu and Heffernan even faster. After Heffernan and Wilken then proposed a set of methods to even more effectively reduce the search space of basic block instances [17] in 2005, Malik, McInnes, and van Beek [7] were able to improve their CP approach to solve the problem also for multiple-issue processors on an even larger set of instances (about $350,000$ basic blocks with up to $2,600$ instructions). While the previous solvers from [9] and [16] could not solve hundreds of these instances to optimality [15], there is only one instance that could not be solved by the CP solver within a time limit of ten minutes of CPU and system time for single-issue processors in our experiments. Notably, in [7], the authors emphasize that their search space reductions are key to the success of their solver.

In this paper, we present the first integer programming approach that is competitive to the CP method of Malik, McInnes and van Beek for single-issue processors. It is also the first IP model that is based on the linear ordering problem. With the exception of scheduling models that employ exactly one general integer ('completion time') variable per instruction (see Sect. 5), it is also the first model whose numbers of variables and constraints is strongly polynomial in the size of the instance. Our corresponding implementation is able to solve all but eleven instances of the mentioned benchmark set to optimality within ten minutes of CPU and system time and is faster on some particular instances. We highlight the most important existing search space reduction techniques that are indeed not specific to CP, and we found that several of them can be improved or extended. We also developed some new reduction ideas and symmetry breaking policies that we also report on. Besides that, we believe that our model can inspire future research for similar scheduling problems with delayed precedence constraints since it offers a comprehensively studied approach to enforce a certain number of instructions to be *between* two other instructions, a way to incorporate processor idle cycles into a model, and several classes of valid inequalities that can be used as cutting planes. Last but not least, our model may be relatively straightforwardly adjusted to deal with different objective functions (e.g., the weighted sum of completion times).

This manuscript is built up as follows. Sect. 2 gives basic definitions and notations necessary to understand the problem and to report on existing (Sect. 3) and novel (Sect. 4) search space reduction techniques. The general challenges in modeling scheduling problems with integer programming is the topic of Sect. 5 while Sect. 6 presents our new proposals to do so. In Sect. 7, additional valid inequalities are presented that can be used as cutting planes with our new models. Sect. 8 highlights the main features of our branch-and-cut implementation that is evaluated in Sect. 9. The presentation closes with our conclusions in Sect. 10.

# 2 Definitions and Notations

## 2.1 Formal problem statement

We formulate the basic block instruction scheduling problem for single-issue processors (ISP) as follows.

**Definition 2.1.** [ISP] *Given a set of uni-cycle instructions $I$ and an acyclic precedence relationship $R \subset I \times I$ along with a latency function $\ell : R \to \mathbb{N}_0$, compute a schedule $\sigma : I \to \mathbb{N}_0$ of the instructions $I$ respecting all the precedence relations $R$ and latencies $\ell$ and whose makespan $M = 1 + \max\{\sigma(i) \mid i \in I\}$ is minimum.*

In the literature, the latencies are often also called *delays* and the problem may also be named *single-machine scheduling under delayed precedence constraints*. Using the widely accepted notation proposed in [18], this problem can be classified as $1|prec(l_{ij}), p_j = 1|C_{max}$ or $1|prec(delays), p_j = 1|C_{max}$.

The notion of latencies used in this paper complies to the one used in some older ones (e.g. [3, 19]) but differs from those used in more recent articles targeting also multiple-issue processors, in particular [9, 16, 7]. For a precedence $(i, j) \in R$ with associated latency $\ell(i, j)$, if $t_i$ is the cycle where instruction $i$ is scheduled, then $j$ can be scheduled earliest in cycle $t_i + \ell(i, j) + 1$ (and not $t_i + \ell(i, j)$).

The definition used in the multiple-issue context stems from the fact that two instructions with a write after read dependency can typically be issued in the same clock cycle since the read will take place before the write in the pipeline. However, for single-issue processors, it leads to the peculiarity that there is no semantic difference between a zero- and a one-latency which is why the older definition is preferred.

In general, the latencies cause clock cycles where, even in an optimal schedule, no ready instruction exists. In this case, the processor is left idle or said to execute a *no-operation instruction (NOP)*.

## 2.2 Predecessors, Successors, Independence

For each precedence relation $(i, j) \in R$, we call $i$ a *predecessor* of $j$ and $j$ a *successor* of $i$. Precedence relations are transitive and we denote the transitive closure of $R$ with $R^*$. If $i$ ($j$) is a predecessor (successor) of $j$ ($i$), we also denote this by $i \prec j$ ($j \succ i$), and if neither $i \prec j$ nor $j \prec i$, then $i$ and $j$ are said to be independent (from each other) and we will sometimes denote this by $i \parallel j$. For ease of reference, we will denote the *immediate* predecessor (successor) set of an instruction $v$ by $P(v)$ ($S(v)$). If we want to refer to the entire (transitive) predecessor (successor) set of an instruction, we will write $P^*(v)$ ($S^*(v)$) instead.

## 2.3 Data-Dependency Graphs

A basic block is usually modeled as a data-dependency DAG $G = (V, A)$ along with a weight function $w : A \to \mathbb{N}_0$ where the vertices $V$ identify the instructions $I$ and there is an arc $(i, j) \in A$ for each $(i, j) \in R$ with weight $w(i, j) = \ell(i, j)$. Each vertex with no predecessor (successor) in $G$ is said to be a *source* (*sink*) of $G$. We will assume that a DAG is normalized to have a single *super source*

$b \in V$ and *super sink* $e \in V$. A super source (sink) has a leaving (entering) arc to each source (from each sink) with zero weight. Fig. 1 shows an example.

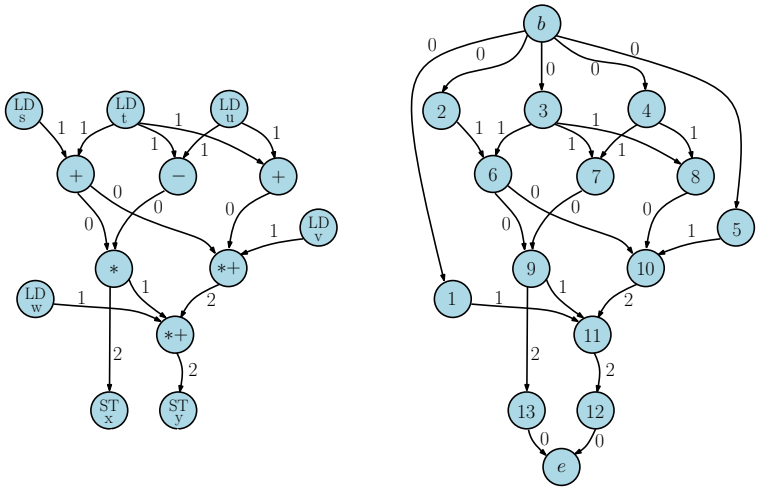


Figure 1: An example data-dependency DAG for a basic block consisting of the computations $x = (s+t)*(u-t)$ and $y = w+(x*(v+((s+t)\cdot(t+u))))$ assuming the presence of multiply-accumulate instructions (left) and its normalized version with a topological numbering of its vertices (right).

In the following, we will treat instructions and their corresponding vertices, precedences and their corresponding arcs as well as latencies and their corresponding arc weights interchangeably. The super source and super sink are interpreted as pseudo-instructions that can be removed from the computed schedule afterwards without altering optimality. Analogously to the precedence relationships $R$, we will denote the transitive closure of the arc set $A$ with $A^*$.

### 2.3.1 Critical Paths and Distances, Transitivity

Let $G = (V, A)$ be a dependency DAG and $i, k \in V$ such that $i \prec k$. Consider a simple path $P = i \to j_1 \to \cdots \to j_p \to k$ from $i$ to $k$ with $p \geq 0$ intermediate vertices in $G$. We refer to $P$'s vertices by $V(P)$ and to its arcs by $A(P)$. The length of $P$ is given by the sum of its arc weights plus the number of intermediate vertices, i.e., by $|V(P)| - 2 + \sum_{(i,j)\in A(P)} w(i,j)$. Any such path $P$ induces a lower bound on the gap between $i$ and $k$ in any feasible schedule. Hence, a longest among such paths (called a *critical path*) between $i$ and $k$ in $G$ imposes the tightest such lower bound, the *critical path distance* $cp(i,k)$.

As we will see, there are several ways to obtain good lower bounds on the minimum number of cycles between two instructions apart from considering paths only. To be unambiguous in notation w.r.t. the critical path distances implied by the given DAG $G$, we introduce a more general and intuitive concept of *distances*, and associate such a distance $d_{i,k}$ with every pair of instructions $i, k \in I, i \neq k$. At each point in time, the distance $d_{i,k}$ reflects the best known lower bound on the gap between $i$ and $k$, regardless how it has been obtained. We make the convention that $d_{i,k}$ is greater than or equal to zero if $i \prec k$ and set to $-\infty$ otherwise. Clearly, it holds that $cp(i,k) \geq \ell(i,k)$ for each $(i,k) \in A$

and $d_{i,k} \geq cp(i,k)$ for all $(i,k) \in A^*$. Naturally, if some particular distance $d_{i,k}$ can be improved, this can be equally exploited to potentially improve transitive distance information as when considering critical paths. In particular, for any path $P$ as described above, $d_{i,k} \geq |V(P)| - 2 + d(P)$ where $d(P)$ is the sum of the distances along the path. More generally, each valid set of distance lower bounds for a given instance can itself be interpreted as a dependency DAG that must have the same optimum makespan as the original DAG.

### 2.3.2 Lower and Upper Bounds

We denote the global lower and upper bounds on the optimum makespan $M^*$ by $M_{lb}$ and $M_{ub}$. Furthermore, we consider lower and upper bounds on the issue cycles of each instruction $i \in I$ and denote them by $lb_i$ and $ub_i$. The interval $[lb_i, ub_i]$ will be referred to as the *(scheduling) range* of an instruction $i \in I$.

We assume the super source $b \in V$ to be assigned clock cycle zero, i.e., $lb_b = ub_b = 0$. For all the instructions $i$, lower bounds on the issue cycles (and thus also an initial $M_{lb}$) can be directly determined from the best known distance of an instruction from $b$, i.e., $lb_i = d_{b,i} + 1$ and $lb_e = M_{lb} - 1 = d_{b,e} + 1$. In contrast to that, upper bounds on the issue cycles are always related to a predetermined global upper bound $M_{ub}$. More precisely, $ub_e = M_{ub} - 1$ and $ub_i = M_{ub} - d_{i,e} - 1$ for any instruction $i \in V \setminus \{b, e\}$. Further, it holds that $lb_k \geq lb_i + d_{i,k} + 1$ and $ub_i \leq ub_k - d_{i,k} - 1$ for each $(i,k) \in A^*$.

### 2.3.3 Regions

The following notion of sub-DAGs called *regions* [9] is useful.

**Definition 2.2.** (Region [9]). *Let $G = (V, A)$ be a dependency DAG and $s, t \in V$ such that there are at least two vertex-disjoint paths between $s$ and $t$. Define $V_{s,t}$ to be the set of vertices reachable on any $s$-$t$-path in $G$ and $A_{s,t}$ as the union of the arcs of all these paths. Then the DAG $G_{s,t} = (V_{s,t}, A_{s,t})$ is called a region of $G$.*

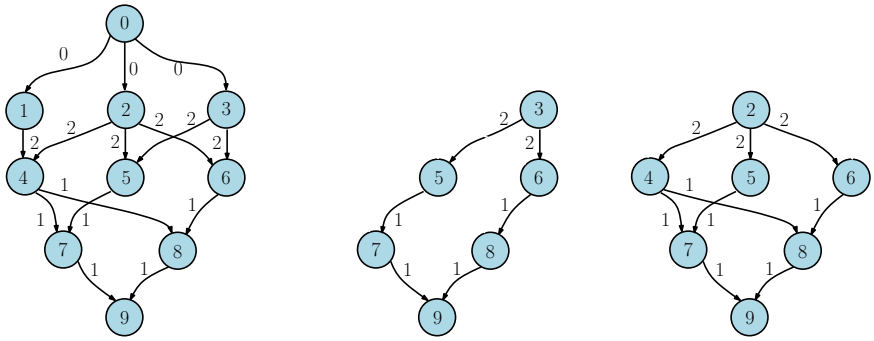


Figure 2: A DAG and two example regions $G_{3,9}$ and $G_{2,9}$.

Regions can be nested. For example, the small region $G_{2,7}$ (which is not shown explicitly) is part of the region $G_{2,9}$ in Fig. 2.

# 3 Search Space Reductions

In this section, we report on already existing search space reduction techniques that we also incorporated into our approach. Further, we were able to slightly improve on some of these methods or to apply them in a different fashion to obtain better results.

Before we start, we provide some central definitions related to global and local lower and upper bounds.

## 3.1 A Relaxation Technique by Rim and Jain

In 1994, Rim and Jain [20] presented a relaxation technique to obtain lower bounds on the makespan of a given dependency DAG. Their method exploits that the upper bounds on issue cycles depend on $M_{ub}$. More generally, for any schedule of length $M$, instruction $i$ must start at time $ub_i^M = M - d_{i,e} - 1$ the latest. Hence, setting $M = M_{lb}$ makes it possible to obtain upper bounds $ub_i^{M_{lb}}$ on issue cycles that must be respected if the schedule length $M_{lb}$ shall be realized. Conversely, if we solve the (relaxed) problem and find an instruction $i$ that is assigned a cycle $c > ub_i^{M_{lb}}$, then $M_{lb}$ can be improved by the respective amount of violation.

The relaxation proposed by Rim and Jain respects only the lower and upper bounds, and the resource constraints (at most one instruction at a time), but neglects the latency constraints. Fortunately, this problem can be solved by a simple greedy algorithm (listed as Algorithm 1) that was also given by the authors.

**Algorithm 1** Greedy Algorithm by Rim and Jain

```
function RimJain(G = (V, A), lb, ub)
    BucketSort(V, ub)          # Sort instructions in increasing order of ub
    g ← 0
    for all v ∈ V do
        Assign v to the earliest free cycle c ≥ lb_v
        if c − ub_v > g then
            g ← c − ub_v
    return g
```

In 1996, Langevin and Cerny [21] proposed to apply this method recursively, namely to each sub-DAG induced by interpreting each predecessor of an instruction $i \in I$ as the sink prior to running the algorithm on the (sub-)DAG with sink $i$. This way, improved lower bounds on the issue cycles of predecessors of $i$ will be already respected when computing a lower bound on $i$'s issue cycle. As the authors report, this leads to better global results in many cases while the runtime observed in practice increases only moderately since many of the considered sub-DAGs are small.

In our method, we go one step further and apply the recursive method to DAGs that reflect the already strengthened distance information that we obtain during our preprocessing.

## 3.2 Improving Bounds on Issue Cycles, Distances and Precedences

### 3.2.1 Lower Bounds on Distances by Smart Counting

Considering regions as defined in Sect. 2.3.3 can help to improve the lower bound on the distance between two dependent instructions in a different way.

**Lemma 3.1.** (from [16]). *Let $G_{s,t} = (V_{s,t}, A_{s,t})$ be a region of a DAG $G = (V, A)$. Then it holds that $d_{s,t} \geq \min\{d_{s,v} \mid (s,v) \in A_{s,t}\} + |V_{s,t}| - 2 + \min\{d_{v,t} \mid (v,t) \in A_{s,t}\}$.*

Correctness of Lemma 3.1 is easy to see: After scheduling the source $s$ of the region, the minimum latency to any successor of $s$ must be respected before any interior vertex can be issued. Then all $|V_{s,t}| - 2$ interior vertices must be scheduled and, between the last one and the sink $t$, again at least the minimum latency must be respected.

Lemma 3.1 is a general result not specific to regions. There is a lower bound technique presented in [22] that exploits similar ideas. In fact, one obtains a lower bound on the distance $d_{i,k}$ between any dependent vertices $i \in V$ and $k \in V$ by determining the intermediate vertices $V_{ik} = \{j \in V \mid i \prec j \text{ and } j \prec k\}$ and computing the value $\min\{d_{i,j} \mid j \in V_{ik}\} + |V_{ik}| + \min\{d_{j,k} \mid j \in V_{ik}\}$.

**Observation 3.2.** *The lower bound on a distance $d_{i,k}$ using the above formula might be improved by ignoring some of the intermediate vertices $V_{ik}$.*

This is true since removing a vertex from the set $V_{ik}$ decreases the term $|V_{ik}|$ by one but may increase one or both of the two computed minima by more than one unit. This poses a new *lower bound optimization problem*:

**Problem 3.3** (Distance Lower Bound Optimization Problem)**.** *Given a DAG $G = (V, A)$ and two vertices $i, k \in V, i \prec k$, compute a set $V^*_{ik} \subseteq \{j \in V \mid i \prec j \text{ and } j \prec k\}$ such that $\min\{d_{i,j} \mid j \in V^*_{ik}\} + |V^*_{ik}| + \min\{d_{j,k} \mid j \in V^*_{ik}\}$ is maximum.*

In its preprocessing phase, the CP solver related to [7] either schedules regions optimally (if they are small) or applies a parameterized algorithm for the above problem to them. To the best of our knowledge, the algorithm is undocumented apart from the openly available source code, but since it is effective and a variant is also incorporated into our solver, we briefly describe it here.

Consider a region $G_{s,t} = (V_{s,t}, A_{s,t})$ and let $W = V_{s,t} \setminus \{s, t\}$. For some parameter $k$, the algorithm calculates the $k$ smallest distances $d_{s,v_1} < d_{s,v_2} < \cdots < d_{s,v_k}$ of vertices $v \in W$ from the source. Further, it computes for each $d_{s,v_i}$ the number $n_{s,i}$ of vertices with strictly smaller distances than $d_{s,v_i}$, i.e., $n_{s,i} = |\{v \in W \mid d_{s,v} < d_{s,v_i}\}|$. The same is done for the $k$ smallest distances $d_{v_1,t} < d_{v_2,t} < \cdots < d_{v_k,t}$ of vertices $v \in W$ to the sink and the respective numbers $n_{t,i}$ of vertices with strictly smaller distances than $d_{v_i,t}$. After that, the index $i \in \{1, \ldots, k\}$ that maximizes $d_{s,v_i} + |W| - n_{s,i}$ is selected. Finally, if it exists, an index $j \in \{1, \ldots, k\}$ is determined such that $|W| - n_{s,i} - n_{t,j} > 0$ and $d_{s,v_i} + d_{v_j,t} + |W| - n_{s,i} - n_{t,j}$ is maximal.

The running time of the algorithm as implemented in the CP solver can be bounded from above by $\mathcal{O}(k\ |V_{s,t}|)$. There, $k$ is set to four. We compute the same regions from the given input DAG but never solve regions exactly

irrespective of their size. Also, by testing all index pairs $i$ and $j$ whether they satisfy $|W| - n_{s,i} - n_{t,j} > 0$, we invest $\mathcal{O}(k^2\ |V_{s,t}|)$ time in the hope to find more regions that can be improved.

### 3.2.2 Improved Bounds on Issue Cycles by Smart Counting

Since lower bounds on issue cycles of instructions are directly related to their distances from the super-source, one can use similar arguments as in the previous subsection to possibly improve them. The following techniques can be analogously applied to upper bounds on issue cycles.

**Lemma 3.4.** (from [16]). *Let $G = (V, A)$ be a DAG and let $v \in V$. Then for any nonempty subset $P' \subseteq P(v)$, it holds that $lb_v \geq \min\{lb_u \mid u \in P'\} + |P'| - 1 + \min\{d_{u,v} \mid u \in P'\} + 1$.*

Correctness can be easily verified by comparing Lemma 3.4 with Lemma 3.1 and replacing $lb_v$ by $d_{b,v} + 1$, $lb_u$ by $d_{b,u} + 1$, and then setting $s = b$ and $t = v$. Again, we may formulate an associated optimization problem.

**Problem 3.5** (Issue-Cycle Lower Bound Optimization Problem)**.** *Given a DAG $G = (V, A)$ and a vertex $v \in V$, compute a set $P^* \subseteq P(v)$ such that $\min\{lb_u \mid u \in P^*\} + |P^*| - 1 + \min\{d_{u,v} \mid u \in P^*\} + 1$ is maximum.*

In the solver related to [7], this problem is tackled by a simple algorithm. It first sorts the predecessors of the vertex $v \in V$ by their lower bounds. Then, for each predecessor $p \in P(v)$, it constructs a lower bound on $v$'s issue cycle by summing up the lower bound of $p$, the number of predecessors with a greater-or-equal lower bound, and the minimum distance to $v$ among these. This algorithm runs in time $\mathcal{O}(|P(v)|^2)$ in the worst case. The underlying methodology is quite similar to another lower bounding technique called *tighter ASAP* presented in [23]. In fact, the algorithm can quite easily be improved by using the distances $d_{p,v}$ of predecessors $p \in P(v)$ as a second criterion to break ties such that whenever multiple predecessors of $v$ have the same lower bound, one with the smallest distance to $v$ will be processed first. This improved version is used within our solver, too.

### 3.2.3 Improved Bounds on Issue Cycles by Interval Considerations

Another method to improve lower bounds on issue cycles and to also remove symmetry from the problem can be derived by considering time intervals. Let $I(a, b)$ be the set of instructions that can possibly be scheduled in the interval $[a, b]$, i.e., $I(a, b) = \{i \mid i \in I,\ lb_i \leq b \text{ and } ub_i \geq a\}$.

**Lemma 3.6.** (from [15]). *If there exists an interval $[a, b]$ such that (i) for all $i \in I(a, b)$ it holds that $ub_i = b$, (ii) for all $i \in I(a, b)$, and for all $s \in S(i)$ it holds that $lb_s - d_{i,s} - 1 \geq b$ and (iii) $|I(a, b)| \leq (b - a + 1)$, then the lower bounds $lb_i$ of all the instructions $i \in I(a, b)$ can be set to $a$.*

A proof of this lemma with case distinctions can be found in [8]. The solver related to [7] uses some fast heuristic tests to check whether there are intervals that satisfy these conditions. In our solver, we only use the routine for upper bounds because the version for lower bounds is in conflict with our new symmetry reduction scheme presented in Sect. 4.4.

Another useful concept in order to improve bounds based on interval considerations are so-called *Hall intervals* having their name from their relation to Philip Hall's marriage theorem proved in 1935 [24].

**Definition 3.7.** (Hall interval [24, 25, 26]). *Let $I^*(a,b)$ be the set of instructions that can be scheduled in the interval $[a,b]$ only, i.e., $I^*(a,b) = \{i \mid i \in I, lb_i \geq a \text{ and } ub_i \leq b\}$. The interval $[a,b]$ is called a Hall interval if $|I^*(a,b)| = b-a+1$.*

Hall intervals are those intervals where there is a known set of instructions $I^*(a,b)$ that must consume all the cycles provided by the interval $[a,b]$. It is easy to see that, if $[a,b]$ is a Hall interval and $i$ is an instruction that is in $I(a,b)$ but not in $I^*(a,b)$, then the interval $[a,b]$ can be removed from the scheduling range of $i$. In particular, if $lb_i \in [a,b]$, then $lb_i$ can be improved to $b+1$. Similarly, if $ub_i \in [a,b]$, then $ub_i$ can be improved to $a-1$.

There exist several algorithms to quickly determine Hall intervals from a given set of scheduling ranges. For example, if $n$ is the number of ranges, the algorithms by Puget [25] and Lopez-Ortiz et al. [27] both find all Hall intervals in time $\mathcal{O}(n \log n)$ and also reduce scheduling ranges accordingly. The algorithm by Lopez-Ortiz et al. is used in the CP solver and we also incorporate it into ours. However, we perform an additional run taking (weak) lower and upper bounds on the position of NOPs into account. The associated idea is that we can also treat an interval like a Hall interval if we know that the number of *instructions and NOPs* that must be placed in it is exactly equal to its size.

### 3.2.4 Obtaining New Precedences by DAG Transformations

Heffernan and Wilken [17] present a set of conditions under which additional arcs (precedences) can be inserted into a DAG without altering the optimal makespan. One of their most effective transformations is based on *sub-DAG isomorphism*. Two graphs $G = (V,E)$ and $H = (W,F)$ are *isomorphic* if $|V| = |W|$, $|E| = |F|$, and there exists a mapping $\phi : V \to W$ such that $(u,v) \in E$ holds if and only if $(\phi(u), \phi(v)) \in F$. For weighted graphs, like our dependency DAGs, we also force the weights on the mapped edges to coincide.

**Theorem 3.1.** ([17]). *Let $G = (V,E)$ and $H = (W,F)$ be two isomorphic sub-DAGs. Say $V = \{v_1, \ldots, v_n\}$ and $W = \{w_1, \ldots, w_n\}$. If $G$ and $H$ are such that for all $i \in \{1, \ldots, n\}$*

- *$v_i$ and $w_i$ are independent,*
- *for each predecessor $p \in P(v_i)$, $p \notin V$, it holds that $l(p, v_i) \leq d_{p,w_i}$,*
- *for each successor $s \in S(w_i)$, $s \notin W$, it holds that $l(w_i, s) \leq d_{v_i,s}$, and*
- *for any arc $(w_i, v_j)$, it holds that $l(w_i, v_j) \leq d_{v_j,w_i}$,*

*then adding zero-latency arcs $(v_i, w_i)$ for all $i \in \{1, \ldots, n\}$ preserves the optimal schedule length of the original DAG.*

The last condition of Theorem 3.1 may appear counter-intuitive, because it argues over arcs that 'cross' from one sub-DAG to the other which should be impossible between two DAGs to be tested for isomorphism. However, as mentioned before, the theorem refers to *induced* sub-DAGs of vertex sets of a

common larger DAG. This way, there might exist such arcs between vertices $V$ and $W$ in the complete DAG and the last condition is then necessary in order to define a safe transformation.

Unfortunately, the detection of isomorphic sub-DAGs is $\mathcal{NP}$-complete [28] (there is a simple and polynomial transformation of the general subgraph isomorphism to the sub-DAG isomorphism problem). However, in practice, a lot of small isomorphic sub-DAGs to which Theorem 3.1 may be applied can be found by some rather simple heuristic tests [8] and the resulting search space reduction justifies the invested computation time in the constraint programming solver [7]. Hence, a similar procedure to the one that is implemented there is used within our solver implementation, too.

# 4 New Results and Search Space Reductions

In addition to the existing search space reductions discussed in Sect. 3, we were able to derive new ways to improve bounds and distances, or to find new precedence relationships. They can be combined with any instruction scheduler.

## 4.1 Exploiting Rim-Jain schedules to further improve bounds

Suppose the lower bounding algorithm by Rim and Jain from Sect. 3.1 is run with global lower bound $M_{lb}$ so that no instruction misses its corresponding deadline $ub_i^{M_{lb}}$, but that there is some instruction $i$ with $lb_i \neq ub_i^{M_{lb}}$ placed exactly at its upper bound. By the construction of the algorithm, instruction $i$ could not be placed earlier, i.e., in the interval $[lb_i, ub_i^{M_{lb}} - 1]$, due to a dense block of instructions that all have a smaller-or-equal upper bound. In some particular cases that are pointed out in the following theorem and Fig. 3, we may exploit such situations in order to improve lower bounds.

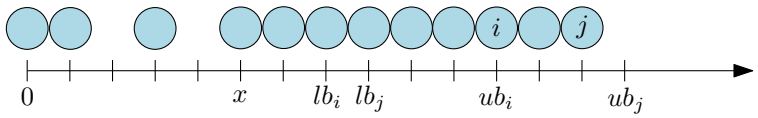


Figure 3: A situation in Rim-Jain schedules that can be exploited to improve lower bounds.

**Theorem 4.1.** *Let $M_{lb}$ be the best lower bound that can be obtained by running the algorithm of Rim and Jain on a DAG $G = (V, A)$. Further, let $\sigma$ be the schedule computed by the algorithm and let $\sigma(v)$ denote the position of each $v \in V$ in $\sigma$. Let $i \in V$ be a vertex with $lb_i < \sigma(i) = ub_i^{M_{lb}}$, and let $[x, ub_i^{M_{lb}}]$ with $x \leq lb_i$ be a dense block of instructions in $\sigma$. If there exists an instruction $j$ such that $ub_j^{M_{lb}} > ub_i^{M_{lb}}$ and $lb_j \in [x, ub_i^{M_{lb}}]$, then the earliest position of $j$ in any feasible schedule of length $M_{lb}$ (if it exists) is $ub_i^{M_{lb}} + 1$.*

*Proof.* Suppose there exists a schedule of length $M_{lb}$ with all lower and upper bounds as given except that $j$'s upper bound is decreased to $ub_i^{M_{lb}}$. Then, given this input, the Rim-Jain algorithm must report a lower bound less or equal to $M_{lb}$. However, while the constructed schedule starts equally to $\sigma$, the algorithm could now break the tie between $i$'s and $j$'s upper bound such that $j$ is processed

before $i$. In this case, $j$ will be placed in a cycle $c \in [lb_j, ub_i^{M_{lb}}]$ with $lb_j \geq x$. This causes all the other instructions in the interval $[c, ub_i^{M_{lb}}]$ of $\sigma$ (including $i$) to be shifted one position to the right. As a consequence, $i$ misses its deadline and the algorithm reports a lower bound of at least $M_{lb}+1$. Hence, no schedule of length $M_{lb}$ can exist if $j$ is enforced to be scheduled at cycle $ub_i^{M_{lb}}$ the latest. Consequently, if a schedule of length $M_{lb}$ does exist, $j$ must be scheduled at cycle $ub_i^{M_{lb}} + 1$ the earliest. □

## 4.2 New precedences due to overlapping intervals

A very simple rule to obtain a new precedence relationship that can however be applied quite frequently in practice is the following.

**Lemma 4.2.** *Let $k \in I$ be an instruction with $ub_k - lb_k = 1$. Suppose now that there exist two instructions $i, j \in I \setminus \{k\}$ such that $ub_i = ub_k$ and $lb_j = lb_k$. Then $i$ must be a predecessor of $j$ in any feasible schedule of $I$.*

*Proof.* The situation associated to this lemma is depicted in Fig. 4. We consider the two cases where instruction $k$ might be scheduled. Suppose $k$ is scheduled in cycle $lb_k$. Then instruction $j$ can be scheduled earliest in cycle $lb_k + 1 = ub_k$. Since $ub_i = ub_k$, it follows immediately, that $i$ must be placed before $k$ and, therefore, also before $j$. In the other case, $k$ is scheduled in cycle $ub_k$, it is clear that instruction $i$ must be scheduled at cycle $ub_k - 1$ the latest. Since $ub_k - 1 = lb_k = lb_j$, it follows that $i$ must precede $j$ also in this case. □

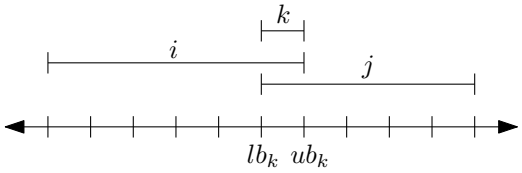


Figure 4: Illustration of the instructions $i$, $j$, and $k$ of Lemma 4.2.

## 4.3 New precedences and bounds due to (Hall) intervals

Let $[a, b]$ be a Hall interval with instruction set $I^*(a, b)$.

**Observation 4.3.** *Let $p \in I \setminus I^*(a, b)$ be a predecessor of one of the instructions $i \in I^*(a, b)$. Then $p$ is a predecessor of* all *the instructions in the set $I^*(a, b)$. The same is true for successors $s \in I \setminus I^*(a, b)$ of any instruction from the set $I^*(a, b)$.*

If a model has a notion of handling NOPs individually, the same observation can be made concerning NOPs. Moreover, one can restrict the number of NOPs between any two instructions $i, j \in I^*(a, b)$ to zero.

Even in the case where an interval $[a, b]$ is not a Hall interval, one can potentially derive some useful restrictions from it that can be expressed as constraints. Again, we consider the associated set of instructions $I^*(a, b)$ that must be scheduled within $[a, b]$ and the following additional sets:

- $I_{\leq} = \{v \in V \setminus I^*(a,b) : ub_v \leq b\}$
- $I_{\geq} = \{v \in V \setminus I^*(a,b) : lb_v \geq a\}$

$I_{\leq}$ is the set of instructions that need to be positioned in cycle $b$ the latest but are not contained in $I^*(a,b)$. Analogously, $I_{\geq}$ is the set of instructions that need to be positioned at cycle $a$ the earliest, but are not contained in $I^*(a,b)$. The idea is now to compute, for each $i \in I^*(a,b)$, an individual upper bound on the number of successors from $I_{\leq}$ and on the number of predecessors from $I_{\geq}$.

**Lemma 4.4.** *Let $[a,b]$, $I^*(a,b)$, $I_{\leq}$ and $I_{\geq}$ be given as above. Let $i \in I^*(a,b)$. The number of successors of $i$ from the set $I_{\leq}$ can be bounded by $\min\{(b-a+1) - |I^*(a,b)|,\ b - lb_i,\ b - a - |S(i) \cap I^*(a,b)|\}$ and the number of predecessors of $i$ from the set $I_{\geq}$ can be bounded by $\min\{(b-a+1) - |I^*(a,b)|,\ ub_i - a,\ b - a - |P(i) \cap I^*(a,b)|\}$.*

*Proof.* Since each $i \in I^*(a,b)$ itself needs to be placed in $[a,b]$, it is clear that any successor $s \notin I^*(a,b)$ of $i$ with upper bound less or equal to $b$ needs to be in the interval as well. The first bound $b - a + 1 - |I^*(a,b)|$ is valid since it is exactly the remaining number of cycles not already occupied by instructions from $I^*(a,b)$. The number of successors is trivially bounded by $b - lb_i$ since $i$ cannot start earlier than in cycle $lb_i$ and hence at most $b - lb_i$ cycles remain afterwards. In the third bound, the number of remaining cycles after placing $i$, which is $b - a$, is reduced by the number of known successors of $i$ from the set $I^*(a,b)$. Summing up, all three bounds are valid and the smallest of them gives the tightest bound on the number of successors from $I_{\leq}$. The proof for the predecessors of $i$ from the set $I_{\geq}$ is analogous. □

## 4.4 Symmetry Breaking with Latest Ready Times

It is common scheduling terminology to call an instruction $i \in I$ *ready* (at cycle $c$), if all of its predecessor instructions have been scheduled and all latency constraints would be satisfied when scheduling $i$ (at cycle $c$). A novel concept to break symmetries considers the *latest* clock cycle where an instruction *must* be ready (or already scheduled) in any case.

**Definition 4.5.** *(Latest Ready Time) Let $i \in I$ be an instruction. The value $lrt_i = \max\{ub_p + \ell(p,i) + 1 \mid p \in P(i)\}$ is called the latest ready time of $i$.*

The latest ready time (LRT) of instruction $i$ is given by the maximal sum of an upper bound of a predecessor instruction $p$ and its latency to $i$ plus one cycle. For artificial predecessors $p$ (those that are not given by the instance but added a posteriori by preprocessing techniques), it is convenient to assume $\ell(p,i) = 0$. Although $\ell(p,i)$ may be only a weak lower bound on the distance between $p$ and $i$ in an optimum schedule, it is guaranteed that instruction $i$ must be *ready* (or already scheduled) at time $lrt_i$ since all predecessors are scheduled, too, and all latencies induced by data dependencies must be satisfied. The following simple but central observation shows up a way to exploit LRTs.

**Observation 4.6.** *Let $\sigma$ be a schedule of the instructions $I$ with makespan $M$. Let $M > |I|$ and suppose that a NOP is placed at cycle $c$. Let $i \in I$ be an instruction with $\sigma(i) > c$ that is however ready at $c$. Then altering $\sigma$ by scheduling $i$ at $c$ leads to a schedule $\sigma'$ with makespan $M' \leq M$.*

Observation 4.6 simply states that, at each clock cycle, it is always optimal to schedule an instruction instead of a NOP if there is at least one ready instruction at hand. This is obvious since both a NOP and an instruction cover one potential delay cycle of instructions issued earlier, but an instruction that is scheduled earlier may release further potential successors earlier whereas scheduling a NOP does not. The resulting schedule must therefore be as least as good as a schedule where a NOP is preponed w.r.t. the ready instruction. As a consequence, we may define the policy that each NOP that shall be placed before instruction $i$ must be placed before $i$ becomes ready, i.e., in particular before cycle $lrt_i$ (cf. Fig. 5). Conversely, if we know that a NOP is placed earliest at a time later than or equal to $lrt_i$, then it could be fixed to be after $i$.

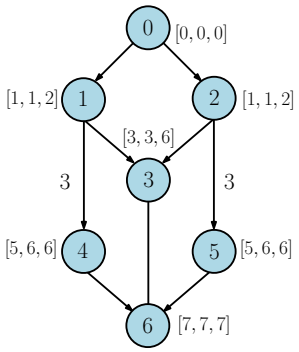


Figure 5: A small instance with labels $[lb_i, lrt_i, ub_i]$ at each instruction, assuming an optimal schedule length of eight cycles. Unlabeled arcs have latency zero. An optimal schedule contains a NOP either at cycle three or four. The upper bound of instruction 3 could be further reduced to 4 since $[5, 6]$ is a Hall interval. The LRT of instruction 3 however indicates that it is ready already at cycle three and can therefore be scheduled before the NOP.

# 5 Modeling Sequential Schedules

The main challenge in modeling sequential scheduling problems mathematically is how to enforce independent instructions to attain different clock cycles. Due to the lack of a 'not equal'-relation this is a difficult task especially in linear programming. If $t_i$ and $t_j$ are integer variables expressing the clock cycles of two independent instructions $i, j \in I$, then it must hold that *either* $t_j \geq t_i + 1$ *or* $t_j \leq t_i - 1$. While these two inequalities are linear expressions, such *disjunctive* [29] constraints are not easy to handle since, at any point in time, only one of them can be enforced and it is subject to the optimization process to find out which one. The equivalent expression $|t_j - t_i| \geq 1$ is not a linear inequality and the feasible solutions to it do not even form a convex set. When using such *completion time variables* $t_i \in \mathbb{N}_0$ for all $i \in I$, a common approach to circumvent this problem is the so-called *big-M* method. In this particular case, one could replace the mentioned two inequalities by $t_j - t_i + Mb_{i,j} \geq 1$ and $t_i - t_j + M(1 - b_{i,j}) \geq 1$ where $M$ is a scalar and $b_{i,j} \in \{0, 1\}$ is a binary variable that controls which of the two inequalities shall be active. For every integral solution one of the inequalities is equal to the original one while the

other one is trivially satisfied due to the choice of $M$. It is clear that such an approach increases the number of variables from linear to quadratic, and that the big-$M$ notation does not yield strong inequalities. For example, for any $M \geq 2$, both constraints are satisfied when setting $t_i = t_j$ and $b_{i,j} = \frac{1}{M}$. So while this formulation works in principle, especially if strong lower and upper bounds for the variables $t_i$ (and $C_{max}$) are known, it cannot be expected to be well-solvable for larger and more difficult instances.

Another common approach is to derive a so-called *time-indexed formulation* (see, e.g., [30]) that does not need a big-$M$ by first computing an upper bound $M_{ub}$ on the makespan such that all potentially necessary clock cycles for an optimum schedule can be expressed as the finite set $T = \{0, \ldots, M_{ub} - 1\}$. This allows for the introduction of decision variables $x_{i,t}$ for each $i \in I$ and all $t \in T$ with the meaning that:

$$x_{i,t} = \begin{cases} 1, & \text{if instruction } i \text{ is scheduled at time } t \\ 0, & \text{otherwise} \end{cases}$$

Although this leads to an entire $\{0,1\}$-IP, there are also some weaknesses. First of all, one can consider such a formulation to be of pseudo-polynomial size since the number of required variables is $|I| \cdot M_{ub}$ with $M_{ub}$ being a numerical value rather than an input size. Further, the linear programming relaxations will typically split instructions over multiple clock cycles and there are several symmetric ways to do this. Nonetheless, Heffernan, Liu, and Wilken [9] were able to schedule a considerable number of basic blocks using a similar model. They also found some cutting planes to separate fractional solutions. Still, it was shown in [16] and [7] that their method is not competitive to the currently best-performing constraint programming methods.

A third way to model the problem comes from the insight that every sequential schedule, independent whether it needs additional NOPs or not, corresponds to a certain *order* or *permutation* of the instructions. A common approach to model permutations is via *linear ordering* variables. Indeed, single machine scheduling with precedences is one of the proposed applications of the linear ordering problem (LOP) mentioned in the corresponding text book [31]. A number of IP formulations based on the LOP have been already proposed, especially in the context of (nondelayed) precedence constrained single machine job-shop scheduling with and without release times and with the objective to minimize the weighted sum of completion times [32, 33, 34, 35, 36, 37, 38, 39, 40]. Even more, for objective functions other than makespan minimization, an interesting comparison of LOP-based formulations to completion time variable and time-indexed formulations has been carried out by Keha et al. [41]. However, as far as this is known to the author, there is so far no publication that successfully applies the LOP for a practical approach to single machine problems with the present form of latencies.

## 5.1 The Linear Ordering Problem

A *linear ordering* of $n$ items $\{1, \ldots, n\}$ is a bijective function $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$, i.e., a ranking, linear sequence, or permutation of the items.

Suppose that for any pair of items $i, j \in \{1, \ldots, n\}, i \neq j$, there is an associated weight (benefit) $c_{i,j}$ that becomes effective if $i$ is ranked before $j$, i.e., if

$\pi(i) < \pi(j)$. Then the *linear ordering problem* (LOP) on $n$ items is the task to find an ordering $\pi^*$ such that $\sum_{i=1}^{n} \sum_{j=i+1}^{n} c_{\pi^*(i),\pi^*(j)}$ is maximum. The LOP is an $\mathcal{NP}$-hard combinatorial optimization problem and has been classified as such by Garey and Johnson [28].

The LOP is often described using a complete directed graph $G_n = (V_n, A_n)$ with arc weights $c_{i,j}$ for each $(i,j) \in A_n$ [31]. So defined, the LOP is to find a subset $T \subsetneq A_n$ such that (i) for every pair of vertices $i$ and $j$ either $(i,j) \in T$ or $(j,i) \in T$, but not both, (ii) $T$ contains no directed cycles, and (iii) $\sum_{(i,j)\in T} c_{i,j}$ is maximum. The associated interpretation is that $\pi(i) < \pi(j)$ holds exactly for the case that $(i,j) \in T$ and vice versa.

Relying on the graph-based model, a common way [42] to formulate the LOP mathematically is to define, for each arc $i, j \in V_n$, $i < j$, a binary variable:

$$x_{i,j} = \begin{cases} 1, \text{ if } \pi(i) < \pi(j) \\ 0, \text{ if } \pi(j) < \pi(i) \end{cases}$$

Then, for $n \geq 3$, the LOP can be stated as an integer program as follows:

$$\begin{aligned} \max \quad & \sum_{i,j\in V_n, i>j} c_{j,i} + \sum_{i,j\in V_n, i<j} (c_{i,j} - c_{j,i})\, x_{i,j} \\ \text{s.t.} \quad & x_{i,j} + x_{j,k} - x_{i,k} \geq 0 && \text{for all } i,j,k \in V_n, i<j<k \quad (1) \\ & x_{i,j} + x_{j,k} - x_{i,k} \leq 1 && \text{for all } i,j,k \in V_n, i<j<k \quad (2) \\ & x_{i,j} \in \{0,1\} && \text{for all } i,j \in V_n, i<j \end{aligned}$$

The objective function maximizes the total weight of the selected arcs. Directed cycles of length two are impossible by construction. Constraints (1) and (2) are the so-called *three-dicycle inequalities* that enforce a solution to have no directed cycles of three or more vertices. In total, the formulation has $\binom{n}{2}$ variables and $2\binom{n}{3}$ nontrivial constraints.

## 5.2 The Linear Ordering Polytope

Let $m = \binom{n}{2}$. For $n \geq 3$, the *linear ordering polytope* can be described as $P_{LO}^n = \text{conv}\{x \in \{0,1\}^m \mid x \text{ satisfies (1) and (2)}\}$ [42]. The three-dicycle inequalities define facets of $P_{LO}^n$ and completely describe it (together with the trivial inequalities) up to $n = 5$. For $n \geq 6$, many more valid and facet-inducing inequalities are known for $P_{LO}^n$ that was intensively studied, e.g, in [42, 43, 44]. However, for many of them the associated separation problem is itself $\mathcal{NP}$-hard [31, 44] and sometimes even no practical separation algorithm is known at all. We refrain from going into further detail here, except for mentioning one of the few exceptional classes of inequalities for later reference that indeed have known polynomial-time separation procedures, namely the so-called $k$-fence inequalities [42].

**Definition 5.1.** ($k$-fence inequalities [42]). *Let $U = \{u_1, \ldots, u_k\}$ and $W = \{w_1, \ldots, w_k\}$ be two disjoint sets of vertices of cardinality $3 \leq k \leq \frac{|V|}{2}$. Then the inequalities*

$$\sum_{i\in\{1,\ldots,k\}} x_{u_i,w_i} + \sum_{i,j\in\{1,\ldots,k\}, i\neq j} x_{w_i,u_j} \leq k^2 - k + 1 \quad (3)$$

*are called $k$-fence inequalities.*

For $n \geq 6$, the $k$-fence inequalities define facets of $P_{LO}^n$ [43]. They are based on particular orientations of a complete bipartite graph $K_{k,k}$.

# 6 New Integer Programming Formulations

In this section, we derive new IP formulations for the instruction scheduling problem based on the LOP. We directly identify the complete directed graph $G_n = (V_n, A_n)$ of the LOP with the vertices of the dependency-DAG $G = (V, A)$, i.e., $V_n = V$. To ease the description, we will assume that all LOP variables w.r.t. the complete graph are present while stating additional constraints based on the arcs of the given dependency-DAG. A strong advantage of linear ordering formulations in modeling precedence-constrained problems is that a known relation $(i, j) \in A^*$ can be immediately exploited by fixing the variable $x_{i,j}$ to one. Besides removing symmetry from the problem, variable fixings reduce the size of the LPs to be solved. This is important, since the number of (integer) variables in the CP approach by Malik et al. is only linear in the number of instructions which is crucial for their success in solving large scale instances with more than $1,000$ instructions. Clearly, this is also one reason why we emphasized on variable fixings in the search space reduction techniques discussed in Sect. 3 and 4. Two major challenges remain: How to formulate latency or, more generally, distance constraints? And how to model NOPs?

## 6.1 Modeling Distances and Betweenness

For a pair if instructions $i, k \in I, i \prec k$, $j$ is between $i$ and $k$ if and only if $j$ is a successor of $i$ and a predecessor of $k$, i.e., if $x_{i,j}x_{j,k} = 1$. This is a quadratic expression that could be linearized, but there is a preferable way to express the same information without the need for additional variables and constraints. Clearly, the product $x_{i,j}x_{j,k}$ is equal to one if and only if the sum $x_{i,j} + x_{j,k}$ is equal to two. The expression $x_{i,j} + x_{j,k} = 2$ is equivalent to $x_{j,k} + (1 - x_{j,i}) = 2$ and therefore to $x_{j,k} - x_{j,i} = 1$, stating that $j$ is between $i$ and $k$, if $j$ is before $k$ but not before $i$.

**Lemma 6.1.** *Let $G = (V, A)$ be a dependency DAG and let an instance of the LOP be defined w.r.t. $G$, i.e., $x_{i,k} = 1$ for all $(i, k) \in A^*$. Let $x$ be an integral solution to the LOP. Then, for each $(i, k) \in A^*$ and each $j \in V \setminus \{i, k\}$, it holds that $x_{j,k} - x_{j,i} \geq 0$.*

*Proof.* Clearly, $x_{j,k} - x_{j,i} \geq -1$. So suppose that this relation holds with equality, since otherwise there is nothing to show. Then $x_{j,k} = 0$ and $x_{i,j} = 0$, and, by assumption, $x_{i,k} = 1$. Hence, the three-dicycle inequality $x_{i,j} + x_{j,k} - x_{i,k} \geq 0$ is violated by $x$ which contradicts the assumption that $x$ is a feasible solution to the LOP. □

Lemma 6.1 shows that we can use the expression $x_{j,k} - x_{j,i}$ to count the instructions between two dependent instructions $i$ and $k$. For now, let us assume that the problem of modeling NOPs is absent, i.e., all required distances between two instructions could be realized with other instructions only. Then it is easy

to see that

$$\sum_{j \in V \setminus \{i,k\}} (x_{j,k} - x_{j,i}) \qquad \geq d_{i,k} \qquad \text{for all } (i,k) \in A \tag{4}$$

is a valid formulation of a distance constraint for $(i,k) \in A$ with distance $d_{i,k}$. It is worth mentioning that, despite the equivalence of the expressions $x_{j,k} - x_{j,i} = 1$ and $x_{i,j} + x_{j,k} = 2$, inequalities (4) are stronger than their apparent pendant inequalities $\sum_{j \in V \setminus \{i,k\}} (x_{i,j} + x_{j,k}) \geq 2d_{i,k}$ because any $j$ *not* between $i$ and $k$ also contributes to either $x_{i,j}$ or $x_{j,k}$ [45].

In the following, we will frequently use the terms *lower bound constraints* and *upper bound constraints* referring to the special distance inequalities where respectively $i$ is the super source $b$ and $k$ is the super sink $e$:

$$\sum_{j \in V \setminus \{b,k\}} x_{j,k} \geq d_{b,k} \qquad \Leftrightarrow \qquad \sum_{j \in V \setminus \{b,k\}} x_{j,k} - \underbrace{\sum_{j \in V \setminus \{b,k\}} x_{j,b}}_{=0} \geq d_{b,k} \tag{5}$$

$$\sum_{j \in V \setminus \{i,e\}} x_{i,j} \geq d_{i,e} \qquad \Leftrightarrow \qquad \underbrace{\sum_{j \in V \setminus \{i,e\}} x_{j,e}}_{=|V|-2} - \sum_{j \in V \setminus \{i,e\}} x_{j,i} \geq d_{i,e} \tag{6}$$

## 6.2 Stronger distance constraints by examination of intermediate instructions

Constraints that enforce distances between pairs of instructions can be strengthened considerably when examining the candidate instructions that may potentially be in between. Let us take a closer look on the just established distance constraints $\sum_{j \in V \setminus \{i,k\}} x_{j,k} - \sum_{j \in V \setminus \{i,k\}} x_{j,i} \geq d_{i,k}$.

The left hand side considers all the instructions $j \in V \setminus \{i,k\}$ while, except for $i = b$ and $k = e$, clearly not all these instructions are indeed candidates in order to attain a position between $i$ and $k$. We will therefore aim at making the left hand side as sparse as possible such that the right hand side imposes a maximal restriction on the real candidate instructions. The following ideas are applicable not only to our but to any model that uses variables permitting to express constraints on the set of instructions placed *between* two other instructions. To constitute a first simple observation, we consider lower bound constraints as special cases of distance constraints.

**Observation 6.2.** *Let $k \in V$ be an instruction with lower bound $lb_k$. Then any instruction $p$ that is placed at some cycle $c \in [0, lb_k - 1]$ must have itself $lb_p \leq c$ since otherwise it could not be placed there.*

So while there are potentially many more instructions (and NOPs) that might be placed before instruction $k$, there is only a reduced candidate set *responsible for establishing the lower bound* of $k$. The same is also true for upper bounds, i.e., an instruction $i$ with upper bound $ub_i$ must have at least $M_{ub} - ub_i - 1$ NOPs or instructions $j$ with upper bound $ub_j > ub_i$ succeeding it. This can be exploited by restricting the variables incorporated into the lower bound constraints (5) to instructions from the set $J^k_{lb} = \{j \in V \setminus \{b\} \mid lb_j < lb_k\}$ and those incorporated into the upper bound constraints (6) to stem from the set $J^i_{ub} = \{j \in V \setminus \{e\} \mid ub_j > ub_i\}$.

A similar observation can be made and exploited for distance constraints between two dependent instructions $i, k \in V$. Let $d_{i,k} > 0$ and consider the set of instructions $J = \{j \in V \setminus \{i,k\} \mid j \not\prec i,\ k \not\prec j,\ lb_j < ub_k \text{ and } ub_j > lb_i\}$. Clearly, these are all the candidates that might be between $i$ and $k$, even if they take their respective extreme positions $lb_i$ and $ub_k$. However, we may again ask for the candidate instructions that are *responsible for establishing the distance* $d_{i,k}$ between $i$ *and* $k$ (short *responsible*). Even for the minimal position $lb_k$ that $k$ can attain, an instruction $j$ that is responsible can only be in the range $[lb_k - d_{i,k}, lb_k - 1]$. So the corresponding candidate set is $J_k = \{j \in V \setminus \{i,k\} \mid j \not\prec i,\ k \not\prec j,\ lb_j < ub_k \text{ and } ub_j \geq lb_k - d_{i,k}\}$. Similarly, even for the maximal position $ub_i$ of $i$, instructions $j$ responsible can only be in the range $[ub_i + 1, ub_i + d_{i,k}]$ such that the corresponding candidate set is $J_i = \{j \in V \setminus \{i,k\} \mid j \not\prec i,\ k \not\prec j,\ lb_j \leq ub_i + d_{i,k} \text{ and } ub_j > lb_i\}$.

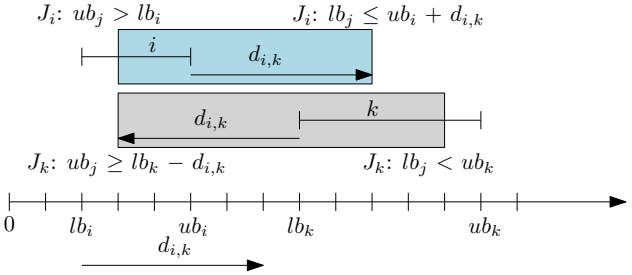


Figure 6: Illustration of the case for general distance relationships $d_{i,k} > 0$.

Fig. 6 illustrates the clock cycle intervals corresponding to $J_i$ and $J_k$ that must be intersected by the scheduling ranges of potentially responsible instructions. Again, the variables to be incorporated into the distance constraints can be reduced accordingly. Moreover, if the candidate sets do not coincide, then it may be beneficial to add two distance constraints related to $J_i$ and $J_k$ for each precedence relationship. We want to elaborate to some more extent when this is the case.

A necessary condition for the sets $J_i$ and $J_k$ to differ and to be smaller than $J$ is that the distance lower bound $d_{i,k}$ must be either not binding for the lower bound of $k$, i.e., $lb_k > lb_i + d_{i,k}$, or not binding for the upper bound of $i$, i.e., $ub_i < ub_k - d_{i,k}$ (or both).

**Theorem 6.3.** *Let $i, k \in V$ be two dependent instructions such that $i \prec k$. If it holds that $lb_k = lb_i + d_{i,k} + 1$ and $ub_i = ub_k - d_{i,k} - 1$, then $J_i = J_k = J$.*

*Proof.* Consider the definition of $J_i = \{j \in V \setminus \{i,k\} \mid j \not\prec i,\ k \not\prec j,\ lb_j \leq ub_i + d_{i,k} \text{ and } ub_j > lb_i\}$. Using the second equation of the theorem, the term $lb_j \leq ub_i + d_{i,k}$ may be replaced by $lb_j \leq ub_k - 1$ which is equal to $lb_j < ub_k$ and, hence, the altered $J_i$ matches exactly the definition of $J$. With the first equation, $J_k$ can equally be turned into $J$. □

In nonbinding cases however, the intersection $J_i \cap J_k$ may even be empty (this is true even in the absence of NOPs while then $|J_i \cup J_k| \geq d_{i,k}$ is a necessary condition for feasibility). Look at Fig. 7. By further reducing $ub_i$ or $d_{i,k}$, or by increasing $lb_k$ in the depicted example, the ranges for the two sets could be made completely disjoint.

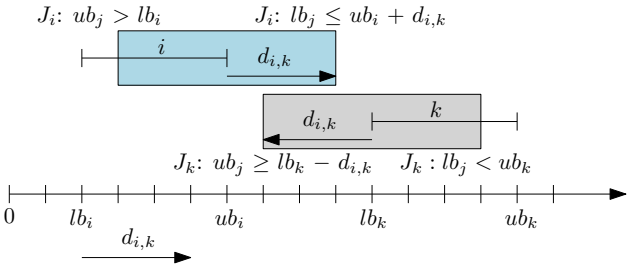


Figure 7: An example where there is only a small overlap of the ranges defining the sets $J_i$ and $J_k$ so that $|J_i \cap J_k|$ might not be large enough to cover $d_{i,k}$.

There are more special cases that allow for an exploitation during the solution process. If the positions of both vertices $i$ and $k$ are fixed, it might still be not clear which particular instructions are the ones to be in between. However, the distance inequality then turns into an equation since we exactly know how many instructions (and NOPs) need to be in between. In this case, all instructions that are known to be *not* in between $i$ and $k$ can be enforced to be either a successor or predecessor of both. Similarly, if the cardinality of a candidate set exactly matches $d_{i,k}$ and it is impossible that NOPs can occur between $i$ and $k$, it is clear that exactly the instructions of the set must be the responsible ones.

## 6.3 Modeling NOPs

We consider two different ways how to incorporate NOPs into the linear ordering model.

The first method considers NOPs as what they in fact are - instructions. With this interpretation, we can simply extend the graph-based LOP formulation (by setting $V = I \cup N$ with $N$ being a set of NOP vertices) and this will already guarantee to obtain a permutation of all instructions and NOPs. The mathematical formulation of this method is given by the basic LOP model extended by inequalities (4) and variable fixings corresponding to the precedences. To minimize the number of NOPs needed, one could, e.g., state the convention that all NOPs that can be placed behind the artificial sink instruction $e \in V$ are proven superfluous and therefore maximize them. The corresponding objective function would be $\max \sum_{n \in N} x_{e,n}$.

The advantages of this approach are clearly that the complete formulation is a $\{0,1\}$-IP and that, for each NOP, we are able to express and iteratively improve the bounds on its position in the schedule like for every instruction. However, since the number $|N|$ of NOP vertices to add depends on (an upper bound on) the optimal schedule length (and therefore on a number rather than an input size), the size of the problem formulation is *pseudo-polynomial* and can become significantly large especially for instances where $|N| > |I|$.

Therefore, we strive to develop a formulation that remains of size $\mathcal{O}(|I|^2)$ in the number of variables and of size $\mathcal{O}(|I|^3)$ in the number of constraints, i.e., is independent from the number of NOPs necessary to construct a feasible schedule. Our method of choice is to have an integer variable $n_i \in \mathbb{N}_0$ that expresses the number of NOPs placed before an instruction $i \in I$. We first state the full model and then proceed with its description and a proof of its correctness.

$$
\begin{aligned}
\min\ & n_e \\
\text{s.t.}\ & x_{i,j} + x_{j,k} - x_{i,k} && \geq 0 && \text{for all } i,j,k \in V, i < j < k \\
& x_{i,j} + x_{j,k} - x_{i,k} && \leq 1 && \text{for all } i,j,k \in V, i < j < k \\
& x_{i,k} && = 1 && \text{for all } (i,k) \in A^* && (7) \\
& (n_k - n_i) + \sum_{j \in V \setminus \{i,k\}} (x_{j,k} - x_{j,i}) && \geq d_{i,k} && \text{for all } (i,k) \in A && (8) \\
& n_k && \geq n_i && \text{for all } (i,k) \in A && (9) \\
& n_k + M_i(1 - x_{i,k}) && \geq n_i && \text{for all } i,k \in V, i \parallel k && (10) \\
& n_i + M_k(1 - x_{k,i}) && \geq n_k && \text{for all } i,k \in V, i \parallel k && (11) \\
& x_{i,j} && \in \{0,1\} && \text{for all } i,j \in V, i < j \\
& n_i && \in \mathbb{N}_0 && \text{for all } i \in V
\end{aligned}
$$

The objective function is to minimize the number of NOPs placed before the artificial sink instruction. Besides the three-di-cycle inequalities from the LOP, we have fixed variables for each precedence $(i,k) \in A^*$. Further, for each $(i,k) \in A$, there is a distance constraint (8) that is composed from constraint (4) by adding the NOPs before $k$ and subtracting the NOPs before $i$, effectively yielding the number of NOPs in between $i$ and $k$. Also, for $(i,k) \in A$, we already know that $n_k \geq n_i$ must hold. For independent instructions $i,k \in V$ however, we would usually need the following two (nonlinear) constraints in order to achieve globally consistent solutions:

$$
\begin{aligned}
n_k &\geq n_i x_{i,k} && \text{for all } i,k \in V, i \parallel k \\
n_i &\geq n_k x_{k,i} && \text{for all } i,k \in V, i \parallel k
\end{aligned}
$$

Here we obtain products of a general integer and a $\{0,1\}$-variable and do not favor linearization by the introduction of additional variables and constraints. Instead, we prefer a linearization using big-$M$ constraints (10) and (11) in this case, since we can hope to compute relatively strong $M$s by using lower and upper bounds $N_i^{lb} \in \mathbb{N}_0$ and $N_i^{ub} \in \mathbb{N}_0$ on each of the NOP variables $n_i$. A good choice for $M_i$ ($M_k$) is an upper bound on the *difference* of NOPs between $k$ and $i$ ($i$ and $k$) in the case that $k$ precedes (succeeds) $i$. Hence, $M_i$ should be equal to $N_i^{ub} - N_k^{lb}$ (or greater) and, similarly, $M_k \geq N_k^{ub} - N_i^{lb}$ is a valid choice. In the case that $k$ precedes $i$, it holds that $x_{i,k} = 0$ and the subtraction of $M_i$ makes inequality (10) trivially satisfied while (11) is binding. The other case is analogous. For later reference, we denote the polytope corresponding to the inequalities of the integer program for a DAG $G = (V,A)$ with $v = |V|$ and $m = \binom{n}{2}$ by $P_{ISP}^G = \text{conv}\{(x,n) \in \{0,1\}^m \times \mathbb{N}_0^v \mid x \in P_{LO}^v \text{ and } (x,n) \text{ satisfies (7)-(11)}\}$. We will write just $P_{ISP}$ whenever we want to refer to the set of feasible solutions of the integer program without relation to a distinct graph instance.

**Theorem 6.4.** *Let $G = (V,A)$ be a dependency DAG, $v = |V|$ and $m = \binom{n}{2}$. Then the set of integral solutions to $P_{ISP}^G$ corresponds exactly to the set of feasible schedules $\sigma$ of $G$.*

*Proof.* Moved to Appendix A. □

The advantages concerning the size of this model come at the cost of some disadvantages. First of all, due to the big-M constraints, it is possible in fractional solutions $(x, n) \in [0,1]^m \times \mathbb{R}_0^n$ that $n_j < n_i x_{i,j}$. In essence, the $n$-variables reintroduce the same disjunctive modeling challenges as with issue-cycle variables (discussed in Sect. 5) for the original problem itself. However, the lower and upper bounds we can compute on $n$-variables are much better than they would be for issue cycles of instructions leading to much smaller M-values. Another weakness is that the position of NOPs is encoded only implicitly, i.e., we cannot easily improve bounds on their positions during the optimization process and exploit them when formulating constraints. However, despite these issues, we found the model with integer NOP variables promising in our experiments.

# 7 Additional Classes of Inequalities

The following classes of inequalities will be presented w.r.t. to the model with integer NOP variables presented in Sect. 6.3. If the model with linear ordering variables only is used, most of the constraints can be adopted by simply replacing terms with $n_i$-variables by terms employing a sum over the respective variables of the at most $N_i^{ub}$ NOPs before an instruction $i$.

## 7.1 Conditional Issue Cycle Bound Constraints

In general, the lower and upper bounds associated to independent pairs of vertices $i, k \in V$ are unrelated to each other. Nevertheless, a particular relative order may impose some restrictions on the positions of $i$ and $k$. Let $lb_i > lb_k$, such that the conditional position $lb_i + 1$ for the case $x_{i,k} = 1$ is a stronger lower bound than $k$'s usual one and is (due to the strict relation) not immediately implied by a combination of a strengthened lower bound constraint for $k$ and $x_{i,k} = 1$ alone. For the same reason, let $ub_k < ub_i$. Then, with $a = lb_i - lb_k$ and $b = ub_i - ub_k$, we may formulate the following inequalities:

$$n_k + \sum_{j \in V \setminus \{i,k\}} x_{j,k} \geq\ lb_k + a x_{i,k} \qquad \text{for all } i,k \in V, i \parallel k, lb_i > lb_k \tag{12}$$

$$n_i + \sum_{j \in V \setminus \{i,k\}} x_{j,i} \leq\ (ub_i - 1) - b x_{i,k} \qquad \text{for all } i,k \in V, i \parallel k, ub_i > ub_k \tag{13}$$

**Theorem 7.1.** *Inequalities (12) and (13) are valid for $P_{ISP}$.*

*Proof.* We restrict ourselves to the lower bound constraint (12) since the proof for the upper bound version is analogous.

First, let $x_{i,k} = 1$. Then $\pi(i) < \pi(k)$, so the number of predecessors of $k$ can be enforced to be at least $lb_i + 1$. In fact, the right hand side of constraint (12) enforces only $lb_k + lb_i - lb_k = lb_i$ predecessors. This is correct however since, in the case $x_{i,k} = 1$, $i$ is also a predecessor that will not be accounted for on the left hand side. Now, let $x_{i,k} = 0$, i.e., $\pi(k) < \pi(i)$. Since then $i$ is not a predecessor of $k$, still $lb_k$ others need to be enforced. □

At this point, it should be shortly noted that the straightforward implication inequality $n_k + \sum_{j \in V \setminus \{i,k\}} x_{j,k} \geq\ lb_i x_{i,k}$ is also valid, but much weaker than inequality (12) for fractional values of $x_{i,k}$. In contrast to that, the upper bound

version $n_i + \sum_{j \in V \setminus \{i,k\}} x_{j,i} \leq \ (ub_k - 1)x_{i,k}$ is not a valid inequality since it imposes invalid restrictions in the case that $x_{i,k} = 0$.

An important property of inequalities (12) and (13) is that, although the bounds $lb_i + 1$ ($ub_k - 1$) are implied by a combination of lower (upper) bound constraints, three-dicycle inequalities and the case $x_{i,k} = 1$ for integer solutions, there exist fractional LP solutions that violate them, i.e., they are nonredundant and can be used as cutting planes.

**Theorem 7.2.** *$P_{ISP}$ has fractional vertex solutions that violate inequalities (12) and (13).*

*Proof.* Moved to Appendix B.1. □

Lower and upper bound inequalities may be strengthened using the concepts from Sect. 6.2. They may be even further strengthened by taking fixed instructions and Hall intervals into account. For instance, if the conditional position $lb_i + 1$ in inequalities (12) is known to be already attained by another instruction, then the conditional lower bound of $k$ for the case $x_{i,k} = 1$ may be increased to the first cycle not already occupied.

## 7.2 Transitivity-driven Conditional Bound Constraints

Constraints similar to the usual conditional bound inequalities can be derived by considering triples of instructions $i, j, k \in V$, $i < j < k$, where *exactly one* of the three associated precedence decisions is already made. Exploiting that transitivity of precedence relationships must hold, even stronger logical implications on the bounds of instructions may be imposed using conditional expressions.

We discuss in detail the case where $i \prec j$ ($x_{i,j} = 1$) is the only decided relation. The transitivity of precedence relations (the corresponding three-dicycle inequalities) w.r.t. $i$, $j$, and $k$ would then be violated if and only if $x_{j,k} = 1$ *and* $x_{i,k} = 0$ at the same time.

By assuming, e.g., $x_{i,k} = 0$, we can therefore conclude that $x_{j,k}$ has to be zero in any feasible schedule, too. The corresponding order is $\pi(k) < \pi(i) < \pi(j)$ so that, in this case, the position of $j$ must be at least $lb_k + 2$ (while $lb_j \geq lb_i + 1$ already holds since $i \prec j$ is already decided). If this imposes a new constraint on $j$, i.e., $lb_k + 2 > lb_j$, then an inequality of the form

$$n_j + \sum_{a \in V \setminus \{j\}} x_{a,j} \geq \ lb_j + (lb_k + 2 - lb_j)x_{k,i} \tag{14}$$

can be added to the problem. A remarkable property of this construction is that the conditional variable $x_{k,i}$ is not related to $j$. Hence, the inequality imposes restrictions on the position of $j$ from decisions made on the relative order of two other instructions. Another valid implication (where this property does not hold anymore) is that $k$ must be placed at position $ub_j - 2$ the latest if $x_{i,k} = 0$ ($x_{k,i} = 1$). This leads to the following inequality:

$$n_k + \sum_{a \in V \setminus \{i,k\}} x_{a,k} \leq \ (ub_k - 1) - (ub_k - (ub_j - 1))x_{k,i} \tag{15}$$

While the correctness of constraint (14) is easy to verify since it is constructed very similarly to constraint (12), the case of constraint (15) needs some formal explanation.

**Theorem 7.3.** *Inequalities (15) are valid for $P_{ISP}$.*

*Proof.* If $x_{k,i} = 0$, the constraint shall not be more restrictive than the usual upper bound constraint for $k$. In this case, $i$ is a predecessor of $k$ that is not counted on the left hand side, so we may enforce only at most $ub_k - 1$ other predecessors.

In the other case that $x_{k,i} = 1$, $i$ is not a predecessor of $k$ and the position of $k$ shall be smaller or equal to $ub_j - 2$. Hence, it is correct to limit the number of predecessors from $V \setminus \{i, k\}$ (and preceding NOPs) by $(ub_k - 1) - (ub_k - (ub_j - 1)) = ub_k - 1 - ub_k + ub_j - 1 = ub_j - 2$. □

As indicated above, for $x_{i,j} = 1$, further implications can be made by assuming $x_{j,k} = 1$. Further, one can consider similar same case distinctions for $x_{i,j} = 0$ and the other possible relations of the variables $x_{j,k}$ and $x_{i,k}$ (some of which are symmetric [45]).

## 7.3 Conditional NOP Constraints

Let $i, k \in I$ be two independent instructions such that the lower bound on the number of NOPs before $i$, $N_i^{lb}$, is strictly larger than the corresponding lower bound $N_k^{lb}$ of $k$.

The inequalities

$$n_k \geq N_k^{lb} + (N_i^{lb} - N_k^{lb})x_{i,k} \qquad \text{for all } i, k \in V, i \parallel k, N_i^{lb} > N_k^{lb} \tag{16}$$

are valid for $P_{ISP}$ because, if $x_{i,k} = 1$, then $k$ is a successor of $i$ and must have at least as many preceding NOPs as $i$, and, if $x_{i,k} = 0$, then an inequality of this form is no more restrictive than the usual variable lower bound associated to $n_k$. In addition to that, the big-$M$ notation used in the constraints (10) and (11) allows for situations where these constraints may be violated. Let us consider inequalities (10) written as

$$n_k \geq n_i - M_i(1 - x_{i,k}) \qquad \text{for all } i, k \in V, i \parallel k$$

and assume that $x_{i,k}$ takes some fractional value. Then, for $M_i > 0$, there will be some positive amount $x = M_i(1 - x_{i,k})$ subtracted from the value of $n_i$ on the right hand side while there will be a positive amount $y = (N_i^{lb} - N_k^{lb})x_{i,k}$ added to $N_k^{lb}$ on the right hand side of inequalities (16). In particular, in any case where $n_i - x < N_k^{lb} + y$, inequalities (16) impose a stronger lower bound on the value of variable $n_k$. To construct a simple example where a solution that is binding w.r.t. the big-$M$ constraints is violated by inequalities (16), assume further that $n_i = N_i^{lb}$. Then

$$\begin{aligned}
& n_i - M_i(1 - x_{i,k}) < N_k^{lb} + (N_i^{lb} - N_k^{lb})x_{i,k} \\
\Leftrightarrow \quad & N_i^{lb} - M_i + M_i x_{i,k} < N_k^{lb} + (N_i^{lb} - N_k^{lb})x_{i,k} \\
\Leftrightarrow \quad & N_k^{lb} + (N_i^{lb} - N_k^{lb}) - M_i + M_i x_{i,k} < N_k^{lb} + (N_i^{lb} - N_k^{lb}) - ((N_i^{lb} - N_k^{lb})(1 - x_{i,k})) \\
\Leftrightarrow \quad & -M_i + M_i x_{i,k} < -(N_i^{lb} - N_k^{lb}) + (N_i^{lb} - N_k^{lb})x_{i,k}
\end{aligned}$$

holds for any $M_i$ such that $M_i > (N_i^{lb} - N_k^{lb})$. This is the usual case in practice since $M_i$ must be chosen such that it is larger than or equal to $N_i^{ub} - N_k^{lb}$ (cf. Sect. 6.3). A weakness of inequalities (16) is however that they are only helpful in the presence of LP solutions where the variables $n_i$ and $n_k$ take values that are close to their lower bounds.

A symmetric version for NOP upper bounds may be formulated as well:

$$n_i \leq N_i^{ub} - (N_i^{ub} - N_k^{ub})x_{i,k} \qquad \text{for all } i,k \in V, i \parallel k, N_k^{ub} < N_i^{ub} \tag{17}$$

## 7.4 Gap Filling Cuts

The following inequalities target the frequent cases where an instruction $i \in I$ has a lower bound on its issue cycle but the set of instructions that will 'fill' these preceding cycles is not completely determined. More formally, we consider instructions $i \in I$ with issue cycle lower bound $lb_i$ and an upper bound on the number of preceding NOPs $N_i^{ub}$ such that $|P^*(i)| + N_i^{ub} < lb_i$. Let $g$ be the corresponding *lower bound gap*, i.e., $g = lb_i - (|P^*(i)| + N_i^{ub})$. By a similar argument as discussed in Sect. 6.2, there must be at least $g$ instructions $j$ currently independent from $i$ that have a lower bound $lb_j < lb_i$.

**Observation 7.4.** *Let $i \in I$ be an instruction with lower bound gap $g > 0$ and let $I_< = \{j \in I \mid j \parallel i \text{ and } lb_j < lb_i\}$. An instruction $p \in I_<$ must be a gap-filling instruction (i.e., must attain a position $\leq lb_i - 1$) if there are strictly less than $g$ other predecessors of $i$ from the set $I_< \setminus \{p\}$.*

In other words, whenever $i$ has less than $g$ predecessors from $I_< \setminus \{p\}$ closing its gap, we can conclude that $p$ is needed to close it. This implication works only in this direction since $p$ may still be gap-filling if $i$ has $g$ or more other predecessors from $I_< \setminus \{p\}$, namely whenever $i$ attains a position strictly later than $lb_i$. Nonetheless, the observation can be used to construct an effective separation scheme. Let $u_p = ub_p - (lb_i - 1)$ be the individual gap between the original upper bound of $p$ and the upper bound attained if $p$ was one of the instructions to fill the gap before $lb_i$. Then we may formulate the following special upper bound constraint for $p$:

$$n_p + \sum_{j \in I \setminus \{p\}} x_{j,p} \leq ub_p - gu_p + u_p\Big(\sum_{j \in I_< \setminus \{p\}} x_{j,i}\Big) \tag{18}$$

**Theorem 7.5.** *Inequalities (18) are valid for $P_{ISP}$.*

*Proof.* We observe first that inequality (18) is equivalent to the usual upper bound constraint for $p$ if $\sum_{j \in I_< \setminus \{p\}} x_{j,i} = g$. In this case, $i$ has exactly $g$ and therefore sufficient predecessors from the set $I_< \setminus \{p\}$ in order to fill the lower bound gap. Thus, no stronger than its usual upper bound may be imposed on $p$. If $\sum_{j \in I_< \setminus \{p\}} x_{j,i} > g$, then constraint (18) is dominated by $p$'s usual upper bound constraint.

The remaining and interesting case is that $\sum_{j \in I_< \setminus \{p\}} x_{j,i} < g$, i.e., there are strictly less than $g$ predecessors from the set $I_< \setminus \{p\}$. Then $p$ must be one of the predecessors of $i$ responsible to close the lower bound gap. However, since we know that at least $g$ predecessors from $I_<$ must exist, we also know that at least $g - 1$ predecessors from $I_< \setminus \{p\}$ must exist. Hence, for each feasible solution, it holds that $g - \sum_{j \in I_< \setminus \{p\}} x_{j,i} \leq 1$ so that at most $u_p$ is subtracted from $p$'s usual lower bound $ub_p$. By construction, $ub_p - u_p = lb_i - 1$ as intended. □

The beneficial property of this construction is that, while we exploit that $g - \sum_{j \in I_< \setminus \{p\}} x_{j,i} \leq 1$ holds for any *integer feasible* solution, LP solutions may have a difference strictly larger than one in general. The higher the infeasibility of an LP solution is w.r.t. this relation, the stronger will be the reduction of $p$'s upper bound, such that these solutions will frequently be cut off by the inequality. But even for fractional solutions where $0 < g - \sum_{j \in I_< \setminus \{p\}} x_{j,i} \leq 1$ holds, the inequality will lead to a (potentially scaled) reduction of $p$'s upper bound.

**Theorem 7.6.** *$P_{ISP}$ has fractional vertex solutions that violate inequalities (18).*

*Proof.* Moved to Appendix B.2. □

A straightforward separation procedure for these constraints can be implemented to have an asymptotic running time of $\mathcal{O}(|I|^2)$ [45]. The inequalities (18) may even be slightly strengthened and the separation procedure can be improved to potentially find more violations. So far, the property to be a predecessor (successor) of $i$ is used as a certificate for instructions to be (not) gap-filling. However, there are other possible certificates. For instance, an instruction $p$ is also proven (not) to be gap-filling if $p$ is a predecessor (successor) of *any* other instruction that has a lower bound greater than or equal to $lb_i$. Let $L = \{r \in I \mid lb_r \geq lb_i\}$. Then we may determine, for each $j \in I_< \setminus \{p\}$, the minimal $x_{j,r}$ such that $r \in L$ and sum up over all these values instead of just over all $x_{j,i}$.

An analogous version of the constraint can be formulated and separated for *upper bound gaps* of an instruction $i$. In this case, potential successor candidates $I_>$ of $i$ are determined and any $s \in I_>$ must have its lower bound increased to $ub_i + 1$ if not enough other potential successors are in fact successors of $i$. The associated inequality is:

$$n_s + \sum_{j \in I \setminus \{s\}} x_{j,s} \geq\ lb_s + gl_s - l_s \Big( \sum_{j \in I_> \setminus \{s\}} x_{i,j} \Big) \tag{19}$$

To close this subsection, we remark that the concept exploited in these inequalities can be further generalized to *interval filling cuts* [45] whose separation is however more time consuming in practice and therefore not considered for our final solver implementation.

## 7.5 Predecessor / Successor Set Constraints

Sometimes it may be beneficial to enforce a certain bound on the number of predecessors or successors out of a *given particular set* of instructions. In a very general fashion with particular predecessor sets $P \subseteq P^*(i)$ and successor sets $S \subseteq S^*(i)$, such upper bounding (lower bounding with '$\geq$' instead of '$\leq$') constraints can be formulated for an instruction $i \in I$ as $\sum_{j \in P} x_{j,i} \leq k$ and $\sum_{j \in S} x_{i,j} \leq k$. Lemma 4.4 in Sect. 4.3 provides an example to straightforwardly construct such inequalities.

## 7.6 Superiority Inequalities

By a *superiority inequality* we mean a very simple constraint of the form $x_{c,d} \geq x_{a,b}$ for some $a, b, c, d \in V$, $a \neq b$ and $c \neq d$, that has the interpretation $x_{c,d} = 1$

whenever $x_{a,b} = 1$, and $x_{a,b} = 0$ whenever $x_{c,d} = 0$. In general, the impact of such inequalities is rather weak. However, if one succeeds in linking rather unrelated variables due to logical implications, these constraints might help to find solutions or detect infeasibility more quickly.

A possible application is strongly related to the ideas presented in Sect. 4.2. There, we were able to derive new precedences from a particular case of overlapping intervals. Here, making weaker assumptions about the relations between the involved instructions, we are at least able to derive a superiority relationship.

**Theorem 7.7.** *Let $u, v \in V$ such that $lb_u \geq ub_v - 1$. Let $i \in V$, $i \parallel u$, and $i \parallel v$. Then it holds that $i$ is a successor of $v$ whenever $i$ is a successor of $u$, and that $i$ is a predecessor of $u$ whenever $i$ is a predecessor of $v$, i.e., $x_{v,i} \geq x_{u,i}$.*

*Proof.* The only case where $u$ can precede $v$ is if $lb_u = ub_v - 1$ and $u$ attains its lower bound position while $v$ attains its upper bound position. In all other cases, $u$ must be a successor of $v$ and then it is clear that, if $i$ succeeds $u$, it must also succeed $v$ and that, if $i$ precedes $v$, it must also precede $u$.

Coming back to the first case: Suppose that $i$ succeeds $u$. Then the position of $i$ is at least $lb_u + 1 \geq ub_v$. As a consequence, $i$ must succeed also $v$. Finally, suppose that $i$ precedes $v$. Then the position of $i$ is at most $ub_v - 1$. Hence, $i$ must also precede $u$ whose position is at least $ub_v - 1$. □

## 7.7 Variable Equality Constraints

A *variable equality constraint* is literally a constraint that enforces the equality of two linear ordering variables, i.e., $x_{c,d} = x_{a,b}$ for some $a, b, c, d \in V$, $a \neq b$ and $c \neq d$. Being very simple, these constraints can have very different modeling semantics besides their intuitive interpretation. For example, they can be used to model an exclusive-or relation. So if an expression like $x_{a,b} + x_{c,d} = 1$ shall be formulated, then this is equivalent to enforcing $x_{a,b} = x_{d,c}$. This can be easily verified by applying the projection relation $x_{c,d} = 1 - x_{d,c}$. Again, we give a practical application.

In Sect. 4.3, we already made the observation that, for any Hall interval $[a, b]$ with instruction set $I^*(a, b)$, any instruction $j \in I \setminus I^*(a, b)$ that is a predecessor (successor) of any instruction $i \in I^*(a, b)$ must be a predecessor (successor) of all the instructions in the set $I^*(a, b)$. Sometimes, we may find instructions $j \in I \setminus I^*(a, b)$ being neither a predecessor nor a successor of any of the instructions in $I^*(a, b)$. It must then hold that $[a, b] \subset [lb_j, ub_j]$ since otherwise we could decide which side of $[a, b]$ is the right one for $j$. In this case, we can still enforce that $j$ must be either before or after all the instructions $I^*(a, b)$ by adding the constraints:

$$x_{j,u} = x_{j,v} \qquad \text{for all } u, v \in I^*(a, b), u \neq v$$

## 7.8 NOP Difference Constraints

Like for the usual distance constraints, we are sometimes in the situation that we can derive a minimum, maximum or even exact difference between the number of NOPs before two dependent instructions $i$ and $k$. For example, by carefully building the normal distance constraints using the strengthening principles presented in Sect. 6.2, we may find out that the distance $d_{i,k}$ can be covered by at

most $b^{ub}_{i,k}$, $b^{ub}_{i,k} < d_{i,k}$, instructions and that therefore at least $N^{lb}_{i,k} = d_{i,k} - b^{ub}_{i,k}$ NOPs are necessary between the two. If all the $b^{ub}_{i,k}$ instructions are already decided to be successors of $i$ and predecessors of $k$, then the usual distance constraint reduces to a minimum NOP difference constraint $n_k - n_i \geq N^{lb}_{i,k}$.

More interesting is the case where the number of instructions known to be fixed between $i$ and $k$, $b^{lb}_{i,k}$, is large so that $(ub_k - lb_i - 1) - b^{lb}_{i,k} < N^{ub}_k - N^{lb}_i$. Then, the lower bound $b^{lb}_{i,k}$ allows to derive a better upper bound on the number of NOPs between $i$ and $k$ as is given by the NOP variable upper and lower bounds. While we cannot improve the variable bounds since we do not know whether $N^{lb}_i$ needs to be increased or $N^{ub}_k$ needs to be decreased, we may add the inequality $n_k - n_i \leq N^{ub}_{i,k}$ with $N^{ub}_{i,k} = (ub_k - lb_i - 1) - b^{lb}_{i,k}$.

An example where we may apply an exact NOP difference inequality of the form $n_k - n_i = N_{i,k}$ can be obtained in the same fashion as with the variable equality constraint for Hall intervals following Sect. 7.7. Since we know for any Hall interval $[a, b]$ that it is completely filled with instructions, we also know that there can be no NOP contained in $[a, b]$. Hence, the number of NOPs preceding or succeeding any pair of instructions $i, j$ contained in $[a, b]$ must be equal.

# 8 The Branch-and-Cut Approach

We present the main ingredients of our Branch-and-Cut solver for the ISP. It solves the IP formulation associated to the polytope $P_{ISP}$ presented in Sect. 6.3. The three-di-cycle inequalities are not added to the IP from the beginning, but separated instead. Distance constraints are added for each distance relationship that is not already implied transitively.

## 8.1 Formulation as a Feasibility Problem

In Sect. 3, we addressed the fact that upper bounds on the issue cycles of instructions are strongly related to the global upper bound $M_{ub}$ on the makespan. We already discussed that we may therefore consider any $M$ in the range $[M_{lb}, M_{ub}]$ in order to obtain the corresponding issue cycle upper bounds $ub^M_i = M - d_{i,e} - 1$ that need to be respected if a schedule of length $M$ shall be realized. A crucial observation is that new precedences $i \prec j$ can be obtained as soon as $ub^M_i \leq lb_j$ holds for some particular $M$. These precedences may in turn lead to additional issue cycle bound improvements and thus to further precedences. Hence, conceptually fixing the current lower bound schedule length and effectively turning the optimization problem into a (series of) feasibility problem(s) can be very profitable w.r.t. search space reductions and in either finding a schedule of the current length or proving that none exists.

## 8.2 Objective Function

Even though we solve feasibility problems, the objective function is not obsolete and can be used to steer the optimization process towards good or optimal solutions. For instance, it can be observed that schedules being one or two cycles better than the currently best known one usually do not deviate too much from each other. Rather, there are some key instructions moved in their position

such that NOPs can be saved. Let $\sigma$ be the best known schedule so far and let $\sigma(i)$ be the position of $i \in V$. We assign the cost coefficient $c_{i,j} = \sigma(i) - \sigma(j)$ to the linear ordering variable $x_{i,j}$ (the NOP variable coefficients are zero). This objective function (to be minimized) has the following properties:

- The coefficients are strictly negative if $\sigma(i) < \sigma(j)$. Hence, setting $x_{i,j} = 1$ and therefore placing $j$ after $i$ again will be rewarded by the objective function.
- The coefficients are strictly positive in the opposite case and with the same effects.
- The reward depends on how far the two instructions were placed apart before. In particular, a large distance between two instructions is considered as a stronger suggestion to keep the order of the two as before.
- No linear ordering variable will have a zero coefficient since no two instructions can have the same position in $\sigma$ (which ensures that we do not add symmetries this way).

## 8.3 Branching Rules

A common standard branching rule selects from a set of candidate variables with LP value closest to 0.5 the one with the highest absolute objective function coefficient (often referred to as *close half expensive*). Such a rule is also provided by ABACUS and it is applied at all subproblems with depth level less than five, in the hope that this helps in finding a solution quickly (if any exists). After that, we first try to apply the following rules in the order of their presentation. If no variable can be found by them, we fall back to the standard rule.

### 8.3.1 Branching on Critical NOP Variables

We consider instructions $i$ that satisfy their lower bound position only due to a fraction of NOPs, i.e. there is a NOP variable $n_i$ such that $\sum_{j \in V \setminus \{i\}} x_{j,i} + n_i \geq lb_i$, but $\sum_{j \in V \setminus \{i\}} x_{j,i} + \lfloor n_i \rfloor < lb_i$. If there is at least one such variable, we select the one that causes the largest violation of the corresponding $lb_i$ when it is reduced to $\lfloor n_i \rfloor$. In the first created subproblem, the upper bound on the variable will be set to $\lfloor n_i \rfloor$ in the hope that it quickly proves infeasible. In the other one, the lower bound of $n_i$ will be set to $\lceil n_i \rceil$.

### 8.3.2 Branching on Contradictory Positions

This branching rule deals with all pairs of instructions $i, k \in V, i \neq k$, such that, considering only the linear ordering variables, $i$ precedes $k$, but considering also the NOP variables, $i$ succeeds $k$ in total. In other words, we look for variables $x_{i,k}$ such that $\sum_{j \in V, j \neq i} x_{j,i} < \sum_{j \in V, j \neq k} x_{j,k}$, but $n_i + \sum_{j \in V, j \neq i} x_{j,i} > n_k + \sum_{j \in V, j \neq k} x_{j,k}$. Such a scenario is possible due to the big-$M$ constraints (10) and (11). Among all variables satisfying the displayed conditions, we select the one that has its LP value closest to one half.

### 8.3.3 Branching on Equal Positions

Here, we relax the condition of the previous rule to also consider variables whose two involved instructions $i$ and $k$ obtain the same position, i.e., $n_i + \sum_{j \in V, j \neq i} x_{j,i} = n_k + \sum_{j \in V, j \neq k} x_{j,k}$.

### 8.3.4 Branching on Illegal Positions

This is another relaxation where we consider pairs $i, k \in V$, $i \neq k$ such that $\sum_{j \in V, j \neq i} x_{j,i} < \sum_{j \in V, j \neq k} x_{j,k}$, but $n_i \geq n_k$ (although this neither leads to equal nor contradictory positions in total).

## 8.4 Propagation at Subproblems

Whenever a linear ordering variable is set during the branch-and-bound procedure, the corresponding decision induces a new precedence in each of the two resulting subproblems. Before the first LP is solved, we propagate the transitive precedences and distance updates that result from the set branching variable. The resulting new distance constraints are added to the LP and those that have become redundant are removed. In addition, some of the preprocessing steps described in Sect. 3 and 4 are carried out in order to potentially further improve on the data or to detect infeasibility of the subproblem.

## 8.5 Cutting Plane Separation Strategy

For the experimental evaluation, we considered two different separation strategies. The first is a minimum configuration, where the separation is mainly restricted to the three-dicycle inequalities (3CYC). Besides these, there are some constraints that are always separated as byproducts of other routines, especially of those dealing with Hall intervals and the addition of distance constraints. These are:

- Variable equality constraints w.r.t. Hall intervals following Sect. 7.7 (VEC).
- NOP difference constraints exploiting the situations mentioned in Sect. 7.8 (NOPD).

The second 'full' configuration activates all the separation routines for the mentioned classes of inequalities in order to permit an evaluation of their impact. The additionally separated inequalities are:

- Three-fence inequalities (heuristically as described in Sect. 8.5.1), if no violated three-dicycle inequalities were found (3FEN).
- Conditional bound constraints from Sect. 7.1 (CND).
- Conditional bound constraints exploiting transitivity implications following Sect. 7.2 (CNDT).
- Conditional NOP constraints as discussed in Sect. 7.3 (CNOP).
- Gap filling cuts as presented in Sect. 7.4 (GAP).

- Predecessor/successor set constraints as described in Sect. 7.5 and based on Lemma 4.4 from Sect. 4.3 (PSB).
- Superior variable inequalities based on overlapping intervals as described in Sect. 7.6 (SVC).

The capital abbreviations in parenthesis will be used in the tables of the evaluation section. The corresponding separation algorithms are, with the exception of the three-fence heuristic, all of straightforward enumerative character. Further, they are all of polynomial time complexity. Irrespective whether the minimum or full separation configuration was used, a branching step is enforced after at most five iterations of interleaved LP solving and separation or if no violated inequality is found.

### 8.5.1 Fences

We implemented a simple three-fence separation heuristic based on the algorithm described in [42]. Our version is as follows: Determine three arcs $x_{u_i,w_i}$, $i \in \{1,2,3\}$, with no endpoint in common and such that $0.3 \leq x_{u_i,w_i} \leq 0.7$. Let $U$ and $W$ be the ordered sets $(u_1,u_2,u_3)$ and $W = (w_1,w_2,w_3)$ respectively. Looking at the definition in Sect. 5.2 again, we see that the other arcs $x_{w_i,u_j}$, $i,j \in \{1,\ldots,3\}, i \neq j$ of the three-fence inequality corresponding to $U$ and $W$ are immediately implied and a violation by the current LP solution can easily be tested.

## 8.6 A Primal Heuristic based on List Scheduling

In most of the cases, the solved LPs will have either fractional variables or violated three-dicycle constraints, or both, i.e., the solution is not feasible for the integer program. In any of these cases, we apply primal heuristics that construct list schedules by employing the current LP solution in order to make decisions. More specifically, we construct two forward and two backward list schedules as follows. The first forward and backward list schedules obey precedences and latencies from the initial dependency DAG. The second ones obey all precedences and distances known in the current subproblem. The priority of each instruction is its distance to the artificial super sink (backward: super source) in terms of the LP solution. Further, since the precedences change with each branching step, two usual critical path list schedules (again, one forward, one backward) are carried out once at each subproblem. If a new incumbent solution is found by any of these list schedules, it is stored, the global upper bound on the makespan is updated and, if it not does not already match the currently assumed schedule length, the optimization process is restarted with an updated objective function (cf. Sect. 8.2). The primal heuristics are key in finding good and optimal schedules quickly.

## 8.7 Implementation with ABACUS

We implemented our integer programming model with the branch-and-cut framework `ABACUS` [46]. We kept all standard parameters, except that we disabled strong branching, told the framework that optimal objective values must be integer, and limited the number of inequalities to add by $100,000$ per iteration.

The LP solver to be employed can be selected from a list of supported ones. For the subsequently printed results, we chose `CPLEX` in version 12.6 [47].

The choice of `ABACUS` permitted us to implement the interleaving of LP solving, the application of cutting planes, and the propagation at the subproblems of the branch-and-bound tree in a straightforward and flexible way. However, in contrast to (commercial) IP solvers that often provide fast heuristic separators, ABACUS does not provide a fine-tuned separation of general cutting planes for integer programs, such as, e.g., $\{0, \frac{1}{2}\}$-cuts [48]. This is a potential disadvantage since it is likely that these would help in detecting infeasibility of some schedule lengths more quickly, especially because the (feasibility) problem to be solved is indeed to prove that a particular polyhedron contains no integer point. Even more, some of the more complex classes of facet defining inequalities of the linear ordering polytope, such as, e.g., the Möbius ladders [43], are in fact special cases of $\{0, \frac{1}{2}\}$-cuts [44] that could possibly be recognized this way.

Despite knowing about these disadvantages, ABACUS was preferred over a commercial IP optimization software because we did not aim at competing with the CP solver by means of black-box tools. Clearly, the sustained performance of the models derived in this chapter could possibly even be better if even more sophisticated integer programming techniques were implemented or by simply profiting from some closed-source implementation tweaks. Using ABACUS, it could be made sure that the results presented in the following section are achieved by means of the models and techniques that arose from the research presented in this thesis only.

# 9 Experimental Evaluation

We evaluated our approach using the same test suite that was used in the paper presenting the optimal CP approach by Malik, McInnes and van Beek [7]. Fortunately, Peter van Beek sent the instances. The set contains even roughly $17,000$ instances more than were used in their experiments and that we now solved in addition. In total, the set comprises $369,861$ pre- and post-register-allocation basic blocks taken from 28 application codes of the SPEC 2000 integer and floating point benchmarks. When referring to particular instances, the prefix `AR` indicates a post-register-allocation block. Solvers presented prior to the mentioned CP solver failed to solve hundreds of instances from this set. In [7], the authors report that they were able to solve all but two instances to optimality for single-issue processors within a time limit of ten minutes of CPU and system time. In our repeated experiments, that were run single-threaded on a Debian Linux system with `g++` 4.7.2 and optimization level `-O2` on an Intel Core i7-3770T processor running at 2.5 GHz and with 32 GB RAM, we found only one instance that timed out with their solver. Within the same time limit, our solver was not able to solve eleven instances using the minimum separation configuration described in Sect. 8.5.

Table 1 categorizes the instances and the computational results w.r.t. the size of the basic blocks. The third column states the number of instances that could be solved by the applied preprocessing techniques only, i.e., by proving optimality of a list schedule without solving an IP. The large numbers reflect the importance of these methods for instruction scheduling while being completely independent from the final (exact) solution approach. While we concentrated

| | | IP | | | | CP [7] | | |
|---|---|---|---|---|---|---|---|---|
| **Size** | **#DAGs** | **Prep** | **>600s** | **>60s** | **>1s** | **>600s** | **>60s** | **>1s** |
| 3 - 5 | 190,726 | 190,726 | | | | | | |
| 6 - 10 | 96,807 | 96,803 | | | | | | |
| 11 - 15 | 33,229 | 33,166 | | | | | | |
| 16 - 25 | 23,994 | 23,903 | | | | | | 1 |
| 26 - 50 | 15,801 | 15,602 | | | 2 | | | |
| 51 - 100 | 5,945 | 5,819 | 3 | 7 | 11 | | | 17 |
| 101 - 250 | 2,956 | 2,851 | 3 | 3 | 13 | | | 16 |
| 251 - 500 | 256 | 235 | 1 | 1 | 17 | | | 38 |
| 501 - 1000 | 105 | 94 | 2 | 3 | 33 | 1 | 2 | 70 |
| 1001 - 2597 | 42 | 33 | 2 | 11 | 33 | | 7 | 41 |
| **total** | 369,861 | 369,231 | 11 | 25 | 109 | 1 | 9 | 183 |

Table 1: Size distribution of the instances, number of instances solved by preprocessing and timeouts for various time limits.

our presentation (and implementation) to those search space reduction techniques that appeared to have at least some influence on the performance of the solver, it is hard to tell which method has which impact in detail. Many factors, such as the decision when to call these methods at all, the order of calling, and the thresholds that define when to stop the surrounding loop have significant, but very different, consequences that depends mainly on the structure of an instance. Especially for very large instances, an iterative application of the methods described in Sect. 3.2.4 to obtain new precedences and their transitive propagation could run for hours before reaching a fixed point where no more precedences can be derived. We consciously refrained from parameterizing these options and employed only a simple rule that stops the derivation of new precedences if less than half of a percent of the precedences obtained in the very first run are obtained in the current iteration. The other routines are not at all steered and run until a fixed point is reached. In contrast to that, the CP solver sets relative time limits for various subroutines based on the size of the instances. A more intensive tuning of parameters could therefore lead to being even more competitive, but a meaningful evaluation of the several possible parameter settings would require a computational study at the length of this paper for itself. It is also unclear what would be a good measure for the impact of a search space reduction method in this setting.

In total, 74 instances more could be scheduled in less than a second compared to the CP solver. However, there are also 16 more instances that needed between one and 60 seconds and ten instances more that could not be solved within a time limit of ten minutes of CPU and system time. Table 2 lists all the instances that the solver was not able to solve within this limit. The optimization was not always stopped exactly after this time (as is shown in the last column) because the time limit was passed to the internal timer provided by ABACUS for the IP solution phase. So if the preprocessing took a long time or multiple IPs were solved, the total time could exceed the limit given for a single IP. Column ILB gives the initial lower bound on the makespan (after the preprocessing phase). PLB denotes the lower bound that the whole solver was able to prove in the time denoted in the last column, and BEST the best solution it could find.

OPT is the optimum makespan (if known) and IUB is the length of the best initially determined list schedule. We remark that the DAGs corresponding to the instances `AR-12061` and `AR-11852` are identical.

| Basic Block | # Instr. | ILB | PLB | OPT | BEST | IUB | Time |
|---|---|---|---|---|---|---|---|
| crafty/AR-2903 | 495 | 876 | 878 | 879 | 879 | 882 | 632 |
| crafty/AR-4661 | 713 | 1,301 | 1,303 | [1,306, 1,307] | 1,308 | 1,310 | 773 |
| fma3d/6261 | 141 | 250 | 250 | 251 | 251 | 251 | 608 |
| fma3d/5417 | 77 | 99 | 101 | 102 | 102 | 107 | 601 |
| fma3d/6916 | 149 | 259 | 259 | 260 | 260 | 260 | 607 |
| fma3d/AR-9459 | 860 | 923 | 925 | 932 | 938 | 1,039 | 784 |
| jpeg/AR-3529 | 1,824 | 3,554 | 3,554 | 3,554 | 3,556 | 3,561 | 922 |
| mesa/AR-11436 | 1,508 | 1,735 | 1,736 | 1,737 | 1,737 | 1,739 | 699 |
| sixtrack/AR-12061 | 87 | 93 | 94 | 95 | 95 | 101 | 1,007 |
| sixtrack/AR-11852 | 87 | 93 | 94 | 95 | 95 | 101 | 1,007 |
| sixtrack/5960 | 195 | 195 | 195 | 198 | 198 | 210 | 602 |

Table 2: Instances not solved within 600 seconds by our solver (AR = after register allocation).

Instance `AR-4661` is the only one that could be solved by neither of the two methods. Hence, the optimum makespan is unknown. However, the list scheduler implemented into the CP solver found a solution of length 1307 and the solver could prove that no schedule with a makespan smaller than 1306 exists. Only in a single case (`AR-3529`), the lower bound proven by the branch-and-cut solver is optimal, but an optimal solution was not found within the time limit. An optimal solution was also not found for the instances `AR-4661` and `AR-9459` but also without having proved that no better schedule can exist. In all the other cases, the optimum solution has been found, but the instances could eventually not be solved because the solver was not able to prove that no better schedule exists.

Disappointingly, this situation does not change when activating the full separation strategy as described in Sect. 8.5. Tables 3 and 4 show statistical data about the number of subproblems and LPs solved as well as the number of separated inequalities for both separation strategies and the timed-out instances. While it is a positive result that a large number of violated inequalities could be found by the various separation routines, these inequalities could not prove essential in determining infeasibility of the respective integer programs. On the contrary, the additional time spent for separation even led to three more timeouts for the instances `fma3d/5416`, `fma3d/6612`, and `vpr/3140`. Only for the instance `5960`, we can observe that the additional inequalities helped to prove at least the nonexistence of a schedule without NOPs more quickly. The three-fence separator described in Sect. 8.5.1 could only seldom find violated inequalities. However, besides its heuristic nature, other reasons for this may be that it was only invoked if no three-dicycle inequality was violated (which is not often the case in the first iterations) and only up to five cutting plane phases were carried out before the next branch takes place (see also Sect. 8.5). The reason for the strategy to branch early is that, due to the large number of generated inequalities, the time spent at one subproblem can be sometimes very exhaustive. Further, a tailing-off strategy is not promising because the artificial objective function is not really indicative. Even more, the impact of

the (transitive) implications of a branching step when re-applying some of the preprocessing techniques was found to be stronger than the impact of the several classes of inequalities in general. The employed branching rules had only a limited impact on the question *how many* instances can or cannot be solved within the time limit, but may have impact on the solvability of single instances. Despite the fact that some rules worked well on some particular instances, none of them proved to be superior for all of them. Changing the rule can lead to one or more of the instances from Table 2 to be solved in less than ten minutes, but cause others not to be. The improvement over the standard close half expensive rule is very small, in fact, the unexceptional application of this rule causes only two more instances to fail within ten minutes.

The tables 3 and 4 also show that for some of the largest instances only a few subproblems and LPs were solved within the time limit. As discussed in Sect. 8.7, the speed of enumeration is a weakness of the implementation that could easily be alleviated if a practical use of these methods was to be considered.

Subsuming, it appears that the relaxation of integrality ruled out to be rather a disadvantage of the IP method compared to the enumerative construction character of a CP solver when it comes to proving that no schedule of a given length can exist for the hardest instances. Especially for instances with a lot of symmetry, an aggressive fixing and propagation of instructions to issue cycles can detect infeasibility of all possible configurations more quickly than the solution of linear programs where it must be proven that no *fractional* solution exists that satisfies all the inequalities. Another difference to the CP solver is that the LP-solution-based branching rules and primal heuristics typically make the branch-and-cut solver more sensitive to the underlying LP solver, as LPs frequently do not have a unique optimum solution. Different solutions, however, may lead to different branching decisions or results of the primal heuristic, potentially with impact on the solution process. In our case, the objective function further depends on the reference schedule used. Nevertheless, the presented method proved to be successful and reliable in practice for a very large range of instances. The quadratic number of variables turned out to be no severe limitation when it comes to the solution of larger instances. On the contrary, large instances with many NOPs could be handled because the model size does not depend on the makespan. The increase in the number of variables compared to the CP approach appeared to be alleviated by the opportunity to fix many of these variables, to profit from transitive precedence propagation, and to formulate tighter distance constraints having some notion of betweenness of instructions and NOPs.

| Basic Block | IPs | SUB | LPs | 3CYC | 3FEN | CND | CNDT | GAP | VEC | SVC | PSB | NOPD | CNOP | Time |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| crafty/AR-2903.txt | 1 | 112 | 366 | 15,726 | | | | | 0 | | | 142 | | 632 |
| crafty/AR-4661.txt | 1 | 15 | 35 | 27,000 | | | | | 0 | | | 699 | | 773 |
| fma3d/6261.txt | 1 | 25,507 | 60,456 | 463,563 | | | | | 0 | | | 0 | | 608 |
| fma3d/5417.txt | 2 | 2,205 | 5,693 | 132,268 | | | | | 6 | | | 12 | | 601 |
| fma3d/6916.txt | 1 | 23,411 | 53,961 | 397,058 | | | | | 2 | | | 0 | | 607 |
| fma3d/AR-9459.txt | 1 | 8 | 9 | 2,007 | | | | | 0 | | | 1,628 | | 784 |
| jpeg/AR-3529.txt | 1 | 42 | 85 | 1,221 | | | | | 0 | | | 17 | | 922 |
| mesa/AR-11436.txt | 1 | 17 | 43 | 3,841 | | | | | 84 | | | 710 | | 699 |
| sixtrack/AR-12061.txt | 2 | 22,207 | 71,521 | 1,554,133 | | | | | 20 | | | 25 | | 1,007 |
| sixtrack/AR-11852.txt | 2 | 22,268 | 71,723 | 1,557,721 | | | | | 20 | | | 25 | | 1,007 |
| sixtrack/5960.txt | 1 | 371 | 940 | 163,691 | | | | | 0 | | | 0 | | 602 |

Table 3: Solution and separation statistics for the instances not solved when using the minimum separation configuration.



| Basic Block | IPs | SUB | LPs | 3CYC | 3FEN | CND | CNDT | GAP | VEC | SVC | PSB | NOPD | CNOP | Time |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| crafty/AR-2903.txt | 1 | 78 | 316 | 21,488 | 0 | 1,287 | 6,438 | 120 | 0 | 8,728 | 54 | 135 | 1,084 | 611 |
| crafty/AR-4661.txt | 1 | 6 | 12 | 9,000 | 0 | 1,913 | 23,379 | 37 | 0 | 47,698 | 6 | 431 | 2,608 | 629 |
| fma3d/6261.txt | 1 | 16,004 | 38,789 | 290,784 | 15 | 11,309 | 15,591 | 1,349 | 4 | 13,676 | 4 | 0 | 0 | 608 |
| fma3d/5417.txt | 2 | 1,863 | 5,591 | 124,842 | 300 | 9,862 | 30,610 | 1,024 | 11 | 43,127 | 100 | 7 | 0 | 603 |
| fma3d/6916.txt | 1 | 14,523 | 34,743 | 251,369 | 27 | 9,782 | 12,669 | 1,991 | 0 | 9,256 | 8 | 0 | 0 | 608 |
| fma3d/AR-9459.txt | 1 | 10 | 15 | 3.194 | 0 | 198 | 1,068 | 0 | 0 | 113 | 0 | 1,628 | 0 | 934 |
| jpeg/AR-3529.txt | 1 | 33 | 78 | 1.066 | 0 | 312 | 848 | 39 | 121 | 583 | 17 | 24 | 133 | 921 |
| mesa/AR-11436.txt | 1 | 4 | 4 | 2,000 | 0 | 364 | 1,565 | 29 | 56 | 3,707 | 0 | 424 | 2,972 | 853 |
| sixtrack/AR-12061.txt | 2 | 19,536 | 62,959 | 1,287,886 | 8,961 | 97,269 | 423,934 | 9,367 | 80 | 355,462 | 402 | 16 | 3 | 985 |
| sixtrack/AR-11852.txt | 2 | 19,538 | 62,965 | 1,287,926 | 8,961 | 97,272 | 423,945 | 9,368 | 80 | 355,479 | 402 | 16 | 3 | 985 |
| sixtrack/5960.txt | 2 | 237 | 816 | 142,622 | 0 | 10,831 | 121,023 | 635 | 0 | 14,078 | 36 | 0 | 0 | 680 |

Table 4: Solution and separation statistics for the instances not solved when using the full separation configuration.

# 10 Conclusion

We presented a complete branch-and-cut approach to the basic-block instruction scheduling problem for single-issue processors. It consists of existing, extended and new preprocessing techniques, a first strongly polynomial-size IP model based on the linear ordering problem with several additional cutting planes, and a final implementation that is shown to be competitive to state-of-the-art constraint programming methods. Our results confirm the previously observed impression that search space reductions are essential in solving a broad range of real-world instances to optimality. Further, they show that these can be successfully combined not only with CP methods but also with IP methods. Whereas previous IP approaches failed to solve hundreds of instances of the 369,861 pre- and post-register-allocation basic blocks used for the evaluation, our methods fails to solve only eleven instances within a time limit of ten minutes. Nevertheless, the currently best CP method is somewhat more robust as it fails to solve only one instance within this time limit and appears to still have some advantages in proving that a schedule of a certain length does *not* exist. A disappointing result is that even the derivation and application of several cutting planes could not prove to be a tool to determine IP infeasibility more quickly in our setting. Nonetheless, the associated inequalities expose a lot of structure of the problem and typical issues that can occur in fractional LP solutions, so they may be of interest for further research.

# Acknowledgments

We gratefully thank Abid Muslim Malik, Jim McInnes and Peter van Beek for making their constraint programming solver source code publicly available and sending us the instances they derived and used for their experiments.

# References

# References

[1] J. L. Hennessy, D. A. Patterson, Computer Architecture: A Quantitative Approach, Elsevier, 2007.

[2] L. Finta, Z. Liu, Single machine scheduling subject to precedence delays, Discrete Applied Mathematics 70 (3) (1996) 247–266. doi:10.1016/0166-218X(96)00110-2.

[3] D. Bernstein, I. Gertner, Scheduling expressions on a pipelined processor with a maximal delay of one cycle, ACM Trans. Program. Lang. Syst. 11 (1989) 57–66. doi:10.1145/59287.59291.

[4] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[5] D. Bernstein, M. Rodeh, I. Gertner, Approximation algorithms for scheduling arithmetic expressions on pipelined machines, J. Algorithms 10 (1989) 120–139. doi:10.1016/0196-6774(89)90027-8.

[6] M. A. Ertl, A. Krall, Optimal instruction scheduling using constraint logic programming, in: J. Maluszyński, M. Wirsing (Eds.), Programming Lang. Implem. and Logic Programming, Vol. 528 of LNCS, Springer, 1991, pp. 75–86.

[7] A. M. Malik, J. McInnes, P. van Beek, Optimal basic block instruction scheduling for multiple-issue processors using constraing programming, Int. J. on Artificial Intelligence Tools 17 (1) (2008) 37–54.

[8] A. M. Malik, Constraint programming techniques for optimal instruction scheduling, Ph.D. thesis, University of Waterloo, Waterloo, Canada (2008).

[9] K. Wilken, J. Liu, M. Heffernan, Optimal instruction scheduling using integer programming, SIGPLAN Not. 35 (2000) 121–133.

[10] H.-C. Chou, C.-P. Chung, An optimal instruction scheduler for superscalar processor, IEEE Trans. on Parallel and Distributed Systems 6 (3) (1995) 303–313.

[11] S. Haga, R. Barua, EPIC instruction scheduling based on optimal approaches, in: In Proc. 1st Annual Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology, 2001, pp. 22–31.

[12] S. Arya, An optimal instruction-scheduling model for a class of vector processors, IEEE Trans. on Computers 34 (11) (1985) 981–995. doi:http://doi.ieeecomputersociety.org/10.1109/TC.1985.1676531.

[13] C.-M. Chang, C.-M. Chen, C.-T. King, Using integer linear programming for instruction scheduling and register allocation in multi-issue processors, Computers & Mathematics with Applications 34 (9) (1997) 1–14.

[14] R. Leupers, P. Marwedel, Time-constrained code compaction for DSP's, IEEE Trans. Very Large Scale Integr. Syst. 5 (1) (1997) 112–122.

[15] A. M. Malik, J. McInnes, P. van Beek, Optimal basic block instruction scheduling for multiple-issue processors using constraing programming, in: Proc. 18th IEEE Intern. Conf. on Tools with Artificial Intelligence (ICTAI '06), IEEE Computer Society, Los Alamitos, CA, USA, 2006, pp. 279–287.

[16] P. van Beek, K. Wilken, Fast optimal instruction scheduling for single-issue processors with arbitrary latencies, in: Proc. of the 7th Intern. Conf. on Principles and Practice of Constraint Programming, CP '01, Springer, 2001, pp. 625–639. doi:647488.726807.

[17] M. Heffernan, K. Wilken, Data-dependency graph transformations for instruction scheduling, J. of Scheduling 8 (5) (2005) 427–451. doi:10.1007/s10951-005-2862-8.

[18] R. L. Graham, E. L. Lawler, J. K. Lenstra, A. H. G. R. Kan, Optimization and approximation in deterministic sequencing and scheduling: a survey, in: P. L. Hammer, E. L. Johnson, B. H. Korte (Eds.), Discrete Optimization II, Proc. of the Advanced Research Institute on Discrete Optimization and Systems Applications of the Systems Science Panel of NATO and of the Discrete Optimization Symp., Vol. 5 of Annals of Discrete Mathematics, Elsevier, 1979, pp. 287–326.

[19] K. V. Palem, B. B. Simons, Scheduling time-critical instructions on RISC machines, ACM Trans. Program. Lang. Syst. 15 (1993) 632–658. doi:10.1145/155183.155190.

[20] M. Rim, R. Jain, Lower-bound performance estimation for the high-level synthesis scheduling problem, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems 13 (1994) 451–458. doi:10.1109/43.275355.

[21] M. Langevin, E. Cerny, A recursive technique for computing lower-bound performance of schedules, ACM Trans. Des. Autom. Electron. Syst. 1 (4) (1996) 443–455. doi:10.1145/238997.239002.

[22] G. Tiruvuri, M. Chung, Estimation of lower bounds in scheduling algorithms for high-level synthesis, ACM Trans. Des. Autom. Electron. Syst. 3 (2) (1998) 162–180.

[23] H. P. Peixoto, M. F. Jacome, A new technique for estimating lower bounds on latency for high level synthesis, in: Proc. of the 10th Great Lakes Symp. on VLSI, GLSVLSI '00, ACM, New York, NY, USA, 2000, pp. 129–132.

[24] P. Hall, On representatives of subsets, Journal of the London Mathematical Society 10 (1) (1935) 26–30.

[25] J.-F. Puget, A fast algorithm for the bound consistency of alldiff constraints, in: Proc. of the 15th National/10th Conf. on Artificial Intelligence/Innovative Applications of Artificial Intelligence, AAAI '98/IAAI '98, American Association for Artificial Intelligence, Menlo Park, CA, USA, 1998, pp. 359–366. doi:295240.295635.

[26] W. J. van Hoeve, The alldifferent constraint: A survey, CoRR cs.PL/0105015.

[27] A. Lopez-Ortiz, C.-G. Quimper, J. Tromp, P. van Beek, A fast and simple algorithm for bounds consistency of the alldifferent constraint, in: Proc. of the 18th Intern. Joint Conf. on Artificial Intelligence, IJCAI'03, Morgan Kaufmann Publishers, Inc., San Francisco, CA, USA, 2003, pp. 245–250.

[28] M. R. Garey, D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman & Co., New York, NY, USA, 1979.

[29] E. Balas, J. K. Lenstra, A. Vazacopoulos, The one-machine problem with delayed precedence constraints and its use in job shop scheduling, Management Science 41 (1) (1995) 94–109. doi:10.1287/mnsc.41.1.94.

[30] J. P. Sousa, Time indexed formulations of non-preemptive single-machine scheduling problems, Ph.D. thesis, Université Catholique de Louvain, Louvain, Belgium (1989).

[31] R. Martí, G. Reinelt, The Linear Ordering Problem, Springer, 2011.

[32] C. N. Potts, An algorithm for the single machine sequencing problem with precedence constraints, in: V. J. Rayward-Smith (Ed.), Combinatorial Optimization II, Vol. 13 of Mathematical Programming Studies, Springer, 1980, pp. 78–87.

[33] F. A. Chudak, D. S. Hochbaum, A half-integral linear programming relaxation for scheduling precedence-constrained jobs on a single machine, Operation Reseach Letters 25 (5) (1999) 199–204.

[34] L. A. Wolsey, Formulating single machine scheduling problems with precedence constraints, in: J. J. Gabszewic, J. F. Richard, L. A. Wolsey (Eds.), Economic Decision-Making: Games, Econometrics and Optimisation, North-Holland, 1990, pp. 473–484.

[35] M. E. Dyer, L. A. Wolsey, Formulating the single machine sequencing problem with release dates as a mixed integer program, Discrete Applied Mathematics 26 (2-3) (1990) 255–270.

[36] J. R. Correa, A. S. Schulz, Single-machine scheduling with precedence constraints, Mathematics of Operations Research 30 (4) (2005) 1005–1021.

[37] G. L. Nemhauser, M. W. P. Savelsbergh, A cutting plane algorithm for the single machine scheduling problem with release times, in: M. Akgül, H. W. Hamacher, S. Tüfekçi (Eds.), Combinatorial Optimization, Vol. 82 of NATO ASI Series, Springer, 1992, pp. 63–83.

[38] A. S. Schulz, Scheduling to minimize total weighted completion time: Performance guarantees of LP-based heuristics and lower bounds, in: W. H. Cunningham, S. T. McCormick, M. Queyranne (Eds.), Integer Programming and Combinatorial Optimization, Vol. 1084 of LNCS, Springer, 1996, pp. 301–315.

[39] L. A. Hall, A. S. Schulz, D. B. Shmoys, J. Wein, Scheduling to minimize average completion time: Off-line and on-line approximation algorithms, Mathematics of Operations Research 22 (3) (1997) 513–544.

[40] F. Margot, M. Queyranne, Y. Wang, Decompositions, network flows, and a precedence constrained single-machine scheduling problem, Operations Research 51 (6) (2003) 981–992.

[41] A. B. Keha, K. Khowala, J. W. Fowler, Mixed integer programming formulations for single machine scheduling problems, Comput. Ind. Eng. 56 (1) (2009) 357–367. doi:10.1016/j.cie.2008.06.008.

[42] M. Grötschel, M. Jünger, G. Reinelt, A cutting plane algorithm for the linear ordering problem, Operations Research 32 (6) (1984) 1195–1220.

[43] M. Grötschel, M. Jünger, G. Reinelt, Facets of the linear ordering polytope, Mathematical Programming 33 (1) (1985) 43–60.

[44] S. Fiorini, Polyhedral combinatorics of order polytopes, Ph.D. thesis, Université Libre de Bruxelles, Brussels, Belgium (2001).

[45] S. Mallach, Exact integer programming approaches to sequential instruction scheduling and offset assignment, Ph.D. thesis, Universität zu Köln, Cologne, Germany (2015).

[46] M. Elf, C. Gutwenger, M. Jünger, G. Rinaldi, Branch-and-cut algorithms for combinatorial optimization and their implementation in ABACUS, in: Comput. Comb. Opt., Optimal or Provably Near-Optimal Solutions, Vol. 2241 of LNCS, Springer, 2001, pp. 157–222. doi:647776.734760.

[47] CPLEX optimization studio version 12.6, Reference manual, IBM ILOG (2013).

[48] A. Caprara, M. Fischetti, $\{0, \frac{1}{2}\}$-Chvátal-Gomory cuts, Mathematical Programming 74 (3) (1996) 221–235. doi:10.1007/BF02592196.

# A Correctness proof of the IP formulation

**Theorem 1.** *Let $G = (V, A)$ be a dependency DAG, $v = |V|$ and $m = \binom{n}{2}$. Then the set of integral solutions to $P^G_{ISP}$ $F^G_{ISP} = \{(x, n) \in \{0, 1\}^m \times \mathbb{N}^v_0 \mid x \in P^v_{LO}$ and $(x, n)$ satisfies (7)-(11)$\}$ corresponds exactly to the set of feasible schedules $\sigma$ of $G$.*

*Proof.* $\Leftarrow$: Suppose a feasible schedule $\sigma$ of $G$ is given. We construct a corresponding solution $(x, n) \in F^G_{ISP}$ as follows. For each pair $i, j \in V$, $i < j$, we set $x_{i,j} = 1$ if $i$ precedes $j$ in $\sigma$, and $x_{i,j} = 0$ otherwise. Clearly, since $\sigma$ imposes a total order on $V$, $x \in P^v_{LO}$. Because $\sigma$ is feasible, $x$ must also satisfy the precedence constraints (7). For each $i \in V$, we set $n_i$ to the number of NOPs preceding $i$ in $\sigma$. Hence, the position of $i$ in $\sigma$ maps exactly to $n_i + \sum_{j \in V \setminus \{i\}} x_{j,i}$. Let $(i, k) \in A^*$. By construction, $d^\sigma_{i,k} = (n_k + \sum_{j \in V \setminus \{k\}} x_{j,k}) - (n_i + \sum_{j \in V \setminus \{i\}} x_{j,i}) - x_{i,k}$. Since $x_{i,k}$ is equal to one and hence contributes only to the first sum, an equivalent expression is $d^\sigma_{i,k} = (n_k - n_i) + \sum_{j \in V \setminus \{i,k\}} (x_{j,k} - x_{j,i})$. Because $\sigma$ is a feasible schedule, $d^\sigma_{i,k} \geq d_{i,k}$ and (8) is satisfied. For the same reason, constraints (9) are satisfied by the $n_i$ variables set as described. If $x_{i,k} = 1$, constraint (10) coincides with (9) and, with the proposed choice of $M_k$, constraint (11) evaluates to $n_i \geq N^{lb}_i + n_k - N^{ub}_k$ which is trivially satisfied. If $x_{i,k} = 0$, the roles of $i$ and $k$ are exchanged which leads to an equally feasible situation.

$\Rightarrow$: Now suppose that $(x, n) \in F^G_{ISP}$ and we are asked to construct a feasible schedule $\sigma$ of $G$. First, we set the position of each $i \in V$ in $\sigma$ to $n_i + \sum_{j \in V \setminus \{i\}} x_{j,i}$. Since $(x, n) \in F^G_{ISP}$ implies $x \in P^v_{LO}$, $x$ imposes a linear ordering on $V$. Thus, for each pair $i, k \in V$, $i \neq k$, it holds that either $\sum_{j \in V \setminus \{i\}} x_{j,i} < \sum_{j \in V \setminus \{k\}} x_{j,k}$ if $x_{i,k} = 1$, or $\sum_{j \in V \setminus \{k\}} x_{j,k} < \sum_{j \in V \setminus \{i\}} x_{j,i}$ if $x_{i,k} = 0$. Due to constraints (9), (10) and (11), $n_k \geq n_i$ if $x_{i,k} = 1$, or $n_i \geq n_k$ if $x_{i,k} = 0$. Summing up, either $i$ strictly precedes $k$ or $k$ strictly precedes $i$ for every pair of vertices $i, k \in V$, $i \neq k$. So each position of an instruction $i$ in $\sigma$ is unique and, due to constraint (8), all distances between dependent instructions are satisfied. Hence, $\sigma$ is a unique feasible schedule of $G$. □

# B Proofs for the cutting plane theorems

## B.1 Conditional Issue Cycle Bound Constraints

We start with the proof of Theorem 7.2, here at first restricted to the nonredundancy of the conditional issue cycle lower bound constraints.

**Theorem 2.** *$P_{ISP}$ has fractional vertex solutions that violate inequalities (12) and (13).*

*Proof.* We prove the claim constructively by showing that there exists a basic feasible solution to the linear programming relaxation of the integer program from Sect. 6.3 that violates inequality (12). This can be done by solving the LP relaxation for a particular instance while maximizing the left hand side of inequality (12) in the objective function. If an optimum solution to this LP has an objective function value larger than the right hand side of (12), then it must correspond to a basic feasible solution that violates the inequality.

Fortunately, the instance that we will use for the proof is small and has a simple structure. It is shown in Fig. 8. Basically, it must only be decided which

of the two orders $2-4-3$ and $3-4-2$ shall be taken. W.l.o.g., for the proof, we consider the problem as a pure feasibility problem, assuming the upper bound on the number of NOPs in the integer program to match the optimum (which is zero). Hence, the NOP variables are all fixed to zero in advance and the goal of the integer program is just to show that a feasible solution exists. The corresponding lower and upper bounds on the issue cycles of the vertices are drawn next to them in Fig. 8.

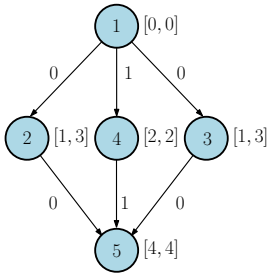


Figure 8: The instance used for the proofs of Theorem 7.2 and Theorem 7.6.

Let us consider vertices 2 and 3. Both have a lower bound of one and are candidates to take the places immediately before or after 4. Any vertex that is a successor of 4 must be placed at a position larger or equal to three. We choose to consider the corresponding conditional lower bound inequality (12) for vertex 3 with $a = lb_4 - lb_3 = 2 - 1 = 1$:

$$n_3 + \sum_{j \in V \setminus \{3,4\}} x_{j,3} \geq lb_3 + 1x_{4,3}$$

$$\Leftrightarrow \quad n_3 + x_{1,3} + x_{2,3} + x_{5,3} \geq lb_3 + x_{4,3}$$

Replacing variables $x_{j,i}$ with $j > i$ by $1 - x_{i,j}$, we obtain the inequality $n_3 + x_{1,3} + x_{2,3} + x_{3,4} + (1 - x_{3,5}) \geq 2$. The only three linear ordering variables that are not fixed in advance are $x_{2,3}$, $x_{2,4}$, and $x_{3,4}$. By inserting the already fixed values, we obtain the inequality $x_{2,3} + x_{3,4} \geq 1$.

Due to its small size, we may write down the linear program completely in Fig. 9, omitting only the trivial inequalities and the already fixed NOP variables. The right LP in Fig. 9 is the reduced form of the left one after fixing also the known values of linear ordering variables.

An optimum vertex solution to this LP, that is in particular a basic feasible solution, is $x_{2,3} = 0$, $x_{2,4} = \frac{1}{2}$, and $x_{3,4} = \frac{1}{2}$. The corresponding binding inequalities are (1b), (6b), and (14) and the objective function value is $x_{2,3} + x_{3,4} = 0 + \frac{1}{2} = \frac{1}{2} < 1$. □

The same LP solution also violates the conditional upper bound constraint (13)

$$
\begin{array}{lrcl}
\min & -1 + x_{1,3} + x_{2,3} + x_{3,4} - x_{3,5} & & \\
\text{s.t.} & & & \\
\text{(1a):} & x_{1,2} + x_{2,3} - x_{1,3} & \leq & 1 \\
\text{(2a):} & x_{1,2} + x_{2,4} - x_{1,4} & \leq & 1 \\
\text{(3a):} & x_{1,2} + x_{2,5} - x_{1,5} & \leq & 1 \\
\text{(4a):} & x_{1,3} + x_{3,4} - x_{1,4} & \leq & 1 \\
\text{(5a):} & x_{1,4} + x_{4,5} - x_{1,5} & \leq & 1 \\
\text{(6a):} & x_{2,3} + x_{3,4} - x_{2,4} & \leq & 1 \\
\text{(7a):} & x_{2,4} + x_{4,5} - x_{2,5} & \leq & 1 \\
\text{(8a)} & x_{3,4} + x_{4,5} - x_{3,5} & \leq & 1 \\
\text{(1b):} & -x_{1,2} - x_{2,3} + x_{1,3} & \leq & 0 \\
\text{(2b):} & -x_{1,2} - x_{2,4} + x_{1,4} & \leq & 0 \\
\text{(3b):} & -x_{1,2} - x_{2,5} + x_{1,5} & \leq & 0 \\
\text{(4b):} & -x_{1,3} - x_{3,4} + x_{1,4} & \leq & 0 \\
\text{(5b):} & -x_{1,4} - x_{4,5} + x_{1,5} & \leq & 0 \\
\text{(6b):} & -x_{2,3} - x_{3,4} + x_{2,4} & \leq & 0 \\
\text{(7b):} & -x_{2,4} - x_{4,5} + x_{2,5} & \leq & 0 \\
\text{(8b):} & -x_{3,4} - x_{4,5} + x_{3,5} & \leq & 0 \\
\text{(9):} & x_{3,2} - x_{3,1} + x_{4,2} - x_{4,1} & \geq & 0 \\
\text{(10):} & x_{2,3} - x_{2,1} + x_{4,3} - x_{4,1} & \geq & 0 \\
\text{(11):} & x_{2,4} - x_{2,1} + x_{3,4} - x_{3,1} & \geq & 1 \\
\text{(12):} & x_{3,5} - x_{3,2} + x_{4,5} - x_{4,2} & \geq & 0 \\
\text{(13):} & x_{2,5} - x_{2,3} + x_{4,5} - x_{4,3} & \geq & 0 \\
\text{(14):} & x_{2,5} - x_{2,4} + x_{3,5} - x_{3,4} & \geq & 1
\end{array}
\qquad
\begin{array}{lrcl}
\min & x_{2,3} + x_{3,4} & & \\
\text{s.t.} & & & \\
 & x_{2,3} & \leq & 1 \\
 & x_{2,4} & \leq & 1 \\
 & 0 & \leq & 0 \\
 & x_{3,4} & \leq & 1 \\
 & 0 & \leq & 0 \\
 & x_{2,3} + x_{3,4} - x_{2,4} & \leq & 1 \\
 & x_{2,4} & \leq & 1 \\
 & x_{3,4} & \leq & 1 \\
 & -x_{2,3} & \leq & 0 \\
 & -x_{2,4} & \leq & 0 \\
 & 0 & \leq & 0 \\
 & -x_{3,4} & \leq & 0 \\
 & 0 & \leq & 0 \\
 & -x_{2,3} - x_{3,4} + x_{2,4} & \leq & 0 \\
 & -x_{2,4} & \leq & 0 \\
 & -x_{3,4} & \leq & 0 \\
 & -x_{2,3} - x_{2,4} & \geq & -2 \\
 & x_{2,3} - x_{3,4} & \geq & -1 \\
 & x_{2,4} + x_{3,4} & \geq & 1 \\
 & x_{2,3} + x_{2,4} & \geq & 0 \\
 & -x_{2,3} + x_{3,4} & \geq & -1 \\
 & -x_{2,4} - x_{3,4} & \geq & -1
\end{array}
$$

Figure 9: The linear program without trivial inequalities and NOP variables (left) and its reduced form after fixing the linear ordering variables (right).

for vertex 2 with $b = ub_2 - ub_4 = 3 - 2 = 1$.

$$
\begin{aligned}
& n_2 + \sum_{j \in V \setminus \{2,4\}} x_{j,2} \leq (ub_2 - 1) - 1x_{2,4} \\
\Leftrightarrow \quad & n_2 + x_{1,2} + x_{3,2} + x_{5,2} \leq (ub_2 - 1) - x_{2,4} \\
\Leftrightarrow \quad & 0 + 1 + (1 - x_{2,3}) + 0 \leq 2 - x_{2,4} \\
\Leftrightarrow \quad & \underbrace{-x_{2,3} + x_{2,4}}_{=\frac{1}{2}} \leq 0
\end{aligned}
$$

## B.2 Gap inequalities

**Theorem 3.** *$P_{ISP}$ has fractional vertex solutions that violate inequalities (18).*

*Proof.* The claim can be proved using the same strategy and instance as for the nonredundancy proof of inequalities (12) from Sect. 7.1.

Vertex 4 in Fig. 8 has a lower bound of two and a lower bound gap $g$ of one since its only decided predecessor is vertex 1 and no NOPs can precede it. The corresponding set $I_<$ consists of the vertices $\{2, 3\}$. For both 2 and 3, the gap $u_p$ is equal to $3 - (2 - 1) = 2$. We choose vertex 3 as our $p \in I_<$. The

corresponding inequality (18) for 3 is

$$
\begin{aligned}
& n_3 + \sum_{j \in I \setminus \{3\}} x_{j,3} - u_3 x_{2,4} \leq u b_3 - g u_3 \\
\Leftrightarrow \quad & n_3 + x_{1,3} + x_{2,3} + x_{4,3} + x_{5,3} - 2x_{2,4} \leq 3 - 1(2) \\
\Leftrightarrow \quad & n_3 + x_{1,3} + x_{2,3} + (1 - x_{3,4}) + (1 - x_{3,5}) - 2x_{2,4} \leq 1 \\
\Leftrightarrow \quad & n_3 + x_{1,3} + x_{2,3} - x_{3,4} - x_{3,5} - 2x_{2,4} \leq -1
\end{aligned}
$$

Inserting the already fixed values, and inverting the inequality to consider it as a minimization objective, we obtain $-1 - x_{2,3} + x_{3,4} + 2x_{2,4} \geq 0$.

Taking a close look at the reduced linear program in the right of Fig. 9, one can see that the value assignments $x_{2,3} = 1$, $x_{2,4} = \frac{1}{2}$, and $x_{3,4} = \frac{1}{2}$ yield another basic feasible solution with binding inequalities (1a), (6a), and (14). The corresponding objective function value is $-1 - x_{2,3} + x_{3,4} + 2x_{2,4} = -1 - 1 + \frac{1}{2} + 2 \cdot \frac{1}{2} = -\frac{1}{2} < 0$. □